Examensarbete
LITH-ITN-ED-EX--07/017--SE

Execution time measurements of processes on the OSE real-time operating system

Liz Malin Ling

2007-09-11

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Science and Technology
Linköpings universitet
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings universitet
601 74 Norrköping

LITH-ITN-ED-EX--07/017--SE

Execution time measurements of processes on the OSE real-time operating system

Examensarbete utfört i Elektronikdesign
vid Linköpings Tekniska Högskola, Campus Norrköping

Liz Malin Ling

Handledare Erik Thorin
Examinator Qin-Zhong Ye

Norrköping 2007-09-11

Avdelning, Institution
Division, Department

Institutionen för teknik och naturvetenskap

Department of Science and Technology

Datum
Date

2007-09-11

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English

☐ _____

Rapporttyp
Report category

Examensarbete
☐ B-uppsats
☐ C-uppsats
☒ D-uppsats

☐ _____

ISBN

ISRN **LITH-ITN-ED-EX--07/017--SE**

Serietitel och serienummer
Title of series, numbering

ISSN

URL för elektronisk version

Titel
Title

Execution time measurements of processes on the OSE real-time operating system

Författare
Author

Liz Malin Ling

Sammanfattning
Abstract

Ett ramverk för kontraktsbaserad schemaläggning och dynamisk resursfördelning i realtidsoperativsystem ska porteras till operativsystemet OSE. Ramverket som utvecklats i ett EU-forskningsprojekt kräver uppmätt process exekveringstid för att fatta riktiga schemaläggningsbeslut. Sådna mätningar görs för närvarande inte i ENEAs RTOS OSE och examensarbetets syfte har därför varit att undersöka möjligheterna att implementera en sådan funktion. Alternativ har hittats och utvärderats, slutligen har ett valts för implementation. Funktionaliteten har verifierats och slutligen har prestanda utvärderats hos den implementerade mätningsmetoden.

Nyckelord
Keyword

RTOS, OSE, scheduling, FRESCOR, ENEA

LINKÖPING UNIVERSITY
ELECTRONIC PRESS

Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Liz Malin Ling

Master Thesis Project Report

Execution Time Measurement Of Processes On The OSE Real-Time Operating System

Liz Malin Ling

A thesis submitted in part fulfilment of the degree of

M.Sc. in Electronic Design Engineering

Moderator: Qin-Zhong Ye (ITN)

Supervisor: Erik Thorin (ENEA)

Linköping University
INSTITUTE OF TECHNOLOGY

ITN, Institution of Technology and Science
Electronic Design Engineering Department

October 4, 2007

Abstract

In a research project by the name FRESCOR, partly funded by the European union, a framework for an RTOS scheduler is being developed. It is designed with a contact based scheduling technique for dynamic resource distribution and it utilizes spare capacities resulting from pessimistic process execution time predictions. As a participant in this project ENEA has agreed to port this scheduling framework to their real-time operating system OSE.

The framework is not designed for any specific RTOS and therefore a framework adaption layer for OSE is designed. An important piece of information requested by the scheduling framework is process execution times which are needed to make correct scheduling decisions. Unfortunately no process execution time measurement functionality is currently available in OSE. Therefore it must be designed within this FRESCOR project.

The objectives of this thesis work has been to find possible methods to perform process execution time measurements, evaluate them and implement one solution. Three methods has been found and evaluated, the RMM implementation method, the kernel handler implementation method and the kernel modification method.

The latter two are very similar and does not differ in measurement accuracy or time resolution. The only advantage that the kernel handler implementation has compared to kernel modification is greater flexibility. Kernel modification requires the development of a new OSE kernel distribution. The kernel handler implementation was also chosen over the RMM implementation since the solution will perform better at high swapping frequencies. The biggest challenge realizing the kernel handler solution is to keep the measurements fast to minimize the increase in context switching latency.

The final implementation was tested to find the extra context switching latency in two different tests. It was determined that, if measurements are performed for both the process being swapped in and for the processes swapped out at a context switch, the mean extra context switch latency is about 14us. The effect from this extra latency on the system performance can not be decided without considering an application.

Assuming a telecom application with a swapping frequency of 1000 swap per second, probably the highest frequency that such an application would have [43], this worst case latency results in 1.4% longer service execution time. However, this is only the case if the processor utilization was initially 100% which is practically never the case.

The conclusion is that for some applications this execution time measurement using kernel handler is probably sufficient. It is reasonable that the extra latency of less than 2% will not cancel the benefit of freeing spare capacity. The latency effect on response time for a specific application and the total execution latency due to swapping frequency should be evaluated. Also the memory required by the application and the memory required to save measurement results should be considered and compared to the memory available.

To conclude, the initial goals of the thesis has been fulfilled. Alternative measurement solutions were found, one solution was chosen for implementation and that solution was designed, finally the implementation was tested and evaluated.

Acknowledgments

Unfamiliar with real-time operating systems I began this thesis work and great challenge six months ago. I am very grateful for the opportunity I got to study and learn in these new areas. For that I would like to thank Ingvar Karlsson and Malcolm Sundberg at ENEA Linköping who granted me this thesis position.

Being new to the area of RTOSs and particularly to the OSE real-time operating system has many times brought the need for assistance. I truly appreciate all the help and patience from ENEA staff and the FRESCOR project group. I would especially like to thank Björn Stenborg and Mathias Bergwall at ENEA Linköping for countless support as well as Magnus Karlsson at ENEA Stockholm for both technical support and good advice.

Finally I would like to thank my instructor at ENEA, Erik Thorin, for all support, guidance and help that he provided me with. It is discussions and information exchange with Erik that has formed this thesis and consequently provided the resulting quality of work. Thank you all for these valuable contributions.

Malin Ling
October 4, 2007

Table of Contents

1 Introduction 5

1.1 Task Definition 5

1.2 Expectations 6

1.3 Method 7

1.4 Delimitations 7

1.5 Thesis Disposition 7

1.6 Background 8

2 Real-Time Operating Systems 11

2.1 Terminology 11

2.2 Issues of Parallel Programming 12

2.3 RTOS Design 14

2.4 Scheduling 17

2.5 Execution time prediction and WCET 18

3 OSE 21

3.1 OSE Architechture 21

4 FRESCOR 23

4.1 FRESCOR Background 23

4.2 Scheduling Based on Contracts 24

4.3 FOSA on OSE 25

5 Execution Time Measurement 27

5.1 Measuring Time in OSE 27

5.2 The RMM Implementation Method 28

5.3 The Kernel Handler Implementation Method 30

5.4 OSE Kernel Modification 31

6 Measurement Method Evaluation 33

6.1 Analysis Approach 33

6.2 Evaluation and Choice 34

7 Implementation 37

7.1 Tools and Environment Configuration . . . 37
7.2 Design Issues . . . 39
7.3 Project Structure Settings . . . 41
7.4 Memory Configuration . . . 41
7.5 Kernel Handlers . . . 43
7.6 Test Application . . . 47
7.7 Software Test Result . . . 48
8 Verification . . . 49
8.1 Hardware Tools and Environment Configuration . . . 49
8.2 Time Overhead Test Specifications . . . 51
8.3 Test Implementation of Test1 . . . 52
8.4 Test Implementation of pBench . . . 54
8.5 Hardware Test Results . . . 56
9 Performance Evaluation . . . 61
9.1 Evaluation Grounds . . . 61
9.2 Memory Utilization . . . 62
9.3 Time Overhead . . . 62
10 Conclusions and Future Work . . . 65
List of Figures . . . 69
List of Tables . . . 71
Bibliography . . . 75
Appendix . . . 76
A krn.con . . . 77
B fosa_ose_exe_time.c . . . 79
C test_app.c . . . 85
D node_info.h . . . 89
E osemain.con . . . 91
F Makefile . . . 93
G fosa_ose_exe_time.mk . . . 95

Chapter 1: **Introduction**

Starting as an aid for industrially specialized embedded applications, real-time operating systems (RTOSs) are now common in a large variety of commercial products. Application areas are for example telecommunications, automotive, defence industry, medical equipment and consumer electronics [40]. A common denominator for these embedded systems are real-time constraints. These systems are often safety critical and must react to the environment instantly on an event. Imagine for example the airbag of a car not going off instantly as a crash occurs, reaction time delay would be disastrous. A real-time operating system provides facilities for implementing a real-time system. It handles the system resources in such way that, if used correctly, meeting deadlines is guaranteed.

ENEA is a Swedish company that has developed such a real-time operating system. It is called OSE which stands for *"Operating System Embedded"*. It is indeed embedded in several systems, "Around half of all telecom radio base stations and 15 percent of all mobile phones in the world have OSE inside", a statement from a press release in the year 2005 [27]. Ericsson and Nokia are two important OSE customers from the telecom industry. SAAB is another OSE customer representing both the automotive and defence application fields. Other customers are Sony, Siemens and Phillips just to mention some [40]. "OSE is a compact, pre-emptive, memory-protected RTOS optimized for communications applications that demand the utmost in reliability, security, and availability" [27].

Continuous development of OSE is performed to meet new system requirements. In a current development project a new scheduling policy for dynamic resource allocation is to be ported to OSE. The scheduling policy is developed from research within the European Union in a project by the name FRESCOR. The scheduling framework which is to be ported to OSE requires some system adaption since the FRESCOR scheduler is not designed for any specific RTOS. As a part of this adaption and system improvement this thesis work has been formulated to investigate the possibilities of process execution time measurement in OSE.

This chapter will present an introductory explanation of the thesis task, what the expectations are on the project, which methods were used to reach the goal and also some limitations. The disposition of the thesis will be declared and the absolutely most trivial basics for comprehension of the thesis will be explained.

1.1 Task Definition

For a real-time operating system to schedule the system resources in a way ensuring that no deadlines are exceeded, the exact deadline must obviously be predefined. In the current development, the OSE operating system is being adjusted to better suit new application areas with less obvious timing constraints. In multimedia systems for instance missing a deadline is not crucial for the system behaviour as in the case with the airbag. Instead exceeding the deadline would merely cause a decreased quality of service. Missing deadlines in such systems can be tolerated up to a certain degree.

In the FRESCOR project a resource allocation scheduling method for such systems, called soft real-time systems, has been designed. FRESCOR stands for *"Framework for Real-time Embedded Systems based on COntRacts"* and implements a technique for dynamic resource allocation [14]. The framework, called FRSH, and applications interface is currently being developed in cooperation between companies and universities throughout Europe. FRSH is not designed to follow any standard such as POSIX, a standard portable operating system interface [28]. Instead it is supposed to be platform independent [18]. ENEA has chosen to take part in this FRESCOR project using OSE, an RTOS of non-POSIX conformance. RT-Linux is another operating system used in the FRESCOR project to which the FRESCOR scheduler will also be ported. RT-Linux is designed according to the POSIX standard. When both operating systems can be used with FRSH, platform independence from POSIX will be demonstrated.

The operating system adaption layer between OSE and the framework needs to provide execution time measurement of OSE processes. This is a parameter essential for the dynamic resource scheduling. Measurement of process execution times has not previously been implemented in OSE and therefore the desired functionality does not currently exist to be provided. The purpose of this thesis is to investigate the possibilities to measure process execution times in OSE. An evaluation of feasible solutions should be performed and one measurement method should be chosen for implementation.

The thesis objectives can be concluded as follows:

- Find and evaluate feasible solutions for execution time measurement.
- Choose with a proper motivation a solution for implementation.
- Implement the solution and verify the measurement functionality.
- Evaluate the quality in performance of the implemented solution.

1.2 Expectations

The thesis aim is to find and analyse possible methods for execution time measurement in OSE. It is reasonable to assume that there are at least one feasible solution. The most appropriate measurement method should be found through careful evaluation and this solution should be implemented and designed for inclusion in the FRSH adaption layer for OSE. A reasonable motivation for implementing the method of choice is to be included in this report as will a detailed description on that implementation. The implementation is expected to be completed after 10 weeks when there is a corresponding milestone in the FRESCOR project plan. The remaining time will be spent testing and evaluating this implementation. Beside writing the thesis report a shorter description of this implementation should also be written for inclusion in a FRESCOR project document.

The resulting conclusions from this work will at least, if the solution is not found to be sufficient for inclusion in the adaption layer, indicate how to find such a solution. The discussions and evaluations of each measurement method as well as the test results from implementation should serve as a guide for future work if needed.

1.3 Method

The thesis project was initially commenced with a study for acquaintance with the scientific area of real-time operating systems. Resource scheduling and the OSE real-time operating system was studied in particular. Discussions on possible methods for execution time measurement in OSE was made with ENEA personnel having great experience in OSE and RTOSs. These discussions lead to a study of the run mode module and the kernel handlers in OSE.

As methods for execution time measurement had been defined parameters to consider at evaluation were specified. With these parameters as a reference the most suitable method for implementation was chosen and designed. To verify the correctness of the measurements the implementation was first tested in the OSE soft core environment. Later to evaluate the quality of the solution hardware tests were performed on a development board from Freescale Semiconductor using a processor with an ARM9 core. Finally these test results were evaluated and conclusions were drawn.

1.4 Delimitations

The project is to be performed as a thesis project submitted for the fulfillment of a M.Sc. degree in electronic design engineering. Therefore the project does not only respond to the deadlines of the FRESCOR project but also to the requirements of the university. As a M.Sc thesis at Linköping University is intended for a duration of twenty weeks the project will be adjusted to include the corresponding amount of work.

This thesis will not include work with any other real-time operating system than OSE nor will it include any study of such operating system. At least one solution to the task should be designed and implemented with the proper motivation. If the solution does not turn out to be the optimal one for inclusion in the adaption layer the thesis work will still be considered completed as twenty weeks has passed. The conclusions from the final tests might suggest possible improvements to the design but implementation of such improvements will not be required unless extra time is available.

The contents of this project should correspond to twenty full time weeks of work at an advancement level appropriate considering previous experience in the real-time operating system area. Writing of the report is included in the twenty weeks time.

1.5 Thesis Disposition

This report will cover the basic information on real-time systems, real-time operating systems and the OSE RTOS. Those basics are necessary for comprehension of the thesis task. The report is written with a student target group of future engineers in mind assuming little or no experience in real-time systems or operating systems. The report is outlined as follows:

- Chapter 1: Thesis introduction, task definition and background.
- Chapter 2: Real-Time Operating Systems basics, scheduling and wcet.
- Chapter 3: Introduction to the RTOS OSE, background and architecture.
- Chapter 4: Information on the FRESCOR project.
- Chapter 5: Presentation of alternative methods for solving the thesis task.
- Chapter 6: Evaluation of the alternative methods and implementation choices.
- Chapter 7: Software implementation, design issues and functionality control.
- Chapter 8: Hardware verification and performance testing.
- Chapter 9: Performance evaluation, analysis of test results.
- Chapter 10: Conclusions and Future Work.

The next section of this chapter, section 1.6, will briefly introduce embedded systems, real-time systems and the concept of real-time. The section is intended for readers unfamiliar with or only briefly acquainted to those fundamental areas forming the ground for RTOS design. Readers with experience in embedded system development are suggested to skip this section. Further more could chapter 2 be left unread by anyone with experience in RTOS design and chapter 3 by any reader already familiar with the OSE real-time operating system. Finally section 7.1 and 8.1 contain detailed descriptions of environment configurations intended as an aid if the development is to be repeated or modified by another ENEA employee.

1.6 Background

Most electronic products today include programmable components controlling peripherals in the system. The programmable component could be a microprocessor, a digital signal processor (DSP) or perhaps a field programmable gate array (FPGA). These programmable devices are often embedded inside the product they are controlling. Characteristics for a system with an embedded programmable device are a predefined purpose, interaction with the environment and limited resources.

Your PC for example is not an embedded system as it is designed with a general purpose processor to perform a variety of tasks. Your keyboard, mouse, router and printer on the other hand are all designed for a specific purpose. A keyboard for example is a small computer, it contains an embedded micro controller that interacts with the environment through the keys. The keyboard has the single purpose of reporting to your PC which keys are being pressed, instantly as they are pressed. A definition of an embedded system can be formulated as follows in definition 1.1.

Definition 1.1 (Embedded System)
An embedded system is a special-purpose computer system which is completely encapsulated by the device it controls. An embedded system has specific requirements and performs predefined tasks, unlike a general purpose personal computer.

Embedded system definitions [30].

Embedded systems are often real-time systems. Real-time embedded systems extend the definition of an embedded system by adding the concept of time as another limited resource. Real-time should not be misinterpreted to "as fast as possible" since executing in real-time really means not exceeding the specified time deadline. Thus, if the deadline is not critical executing as fast as possible is not required. A real-time system must guarantee that deadlines are met and to do so "a real-time application requires a program to respond to stimuli within some small upper limit of response time" [39]. A real-time can be defined as in definition 1.2.

Definition 1.2 (Real-Time System)
A system where correctness depends not only on the correctness of the logical result of the computation, but also on the result delivery time. It is system that responds in a timely, predictable way to unpredictable external stimuli arrivals.

Raquel S. Whittlesey-Harris presentation on real-time operating systems [36].

Real-time applications are often safety critical but not always. This has brought the concepts of hard real-time and soft real-time. In cases where missing a deadline makes the application useless it is called hard real-time. Such applications are particularly safety critical systems as a pacemaker, a flight control system or an airbag that could cause death at malfunction. It could also be sensitive systems such as a car engine control system that at malfunction could cause damage to the engine.

In soft real-time systems missing a deadline can be tolerated to certain extent. This is when missing a deadline merely causes a decrease in quality rather than being crucial to the functionality. Examples of soft real-time systems are multimedia applications as DVD players or mobile phones.

Definition 1.3 (Hard and Soft Real-Time)

- **Hard Real-Time:** The completion of an operation after a hard deadline is considered fatal - ultimately, this may lead to a critical failure of the complete system.
- **Soft Real-Time:** The completion of an operation after a soft deadline is undesirable but not fatal - this merely lead to a decrease in quality.

Subhashis Banerjee, Indian Institute of Technology [41].

A real-time system can be composed of several real-time programs and many real-time embedded systems can be composed to form a cluster. The real-time system must guarantee that all real-time programs in these systems will be executed without exceeding their deadlines. Such a large system is usually composed of both hard and soft real time components and there must be some sort of priority system for distributing the system resources. There are techniques for this such as the foreground/background method, read more about it in the slides by Sundar Resan [34], but you may choose to utilize a real-time operating system for simplification instead.

The advantages when using an RTOS compared to using the foreground/background technique are efficient resource scheduling, the provided abstraction layer from hardware and the support for several hardware configuration and communication protocols. The RTOS facilitates development of embedded systems with parallel activities through an applications interface. This is useful where there are many cooperating embedded real-time systems as in a car for example. Embedded systems control the airbag, the ABS breaks, the fuel system, the entertainment system and the navigation system. Together these systems form a cluster and

obviously, making them all cooperate is a great challenge. In this case one or several RTOSs can be used to control the systems together, simplifying the development. A distributed RTOS such as OSE is particularly useful in systems with more than one microprocessor, as in the car example.

Chapter 2: Real-Time Operating Systems

A real-time operating system is an operating system designed to facilitate the development and utilization of an embedded multitasking real-time system. Particularly in complex systems with many concurrent processes to handle are RTOSs useful. The main task of an RTOS is to allocate the system resources in such a way that all processes will meet their deadlines, hard or soft respectively. Beside efficient resource scheduling algorithms an RTOS also provides a hardware abstraction layer which is beneficial reducing the application programming complexity.

This chapter will focus on real-time operating system concepts and terminology. The RTOS structure and design will be explained and the fundamental issues of the underlaying parallel programming techniques will be discussed. RTOS design issues and race conditions will be mentioned and resource scheduling will particularly be emphasized. Finally process execution time prediction will be mentioned briefly as will the concept of worst case execution time.

2.1 Terminology

Real-time operating system design is a large scientific field and before the fundamentals can be explained the RTOS terminology must be familiar. In this section common RTOS concepts will be explained briefly, please return to this section later if a reminder is needed. Fundamental concepts from the area of parallel programming will be described initially followed by useful concepts for RTOS design discussions.

- **Process**
  A process is piece of program code. It owns a virtual memory address space and has a state defined by register and memory values.

- **Task**
  A task is a set of processes with data dependencies between them.

- **Thread**
  A thread is a part of a process sometimes referred to as a lightweight process. A set of threads constitutes a process.

In a **multitasking** system processes, tasks and threads are executed in a way that appears to provide simultaneous execution. When there is only one processing unit in the system only pseudo parallelism can really be achieved. This is done through **preemption** of processes. Basically preemption means that only a small piece of a processes, task or thread is executed before dispatch. The process is dispatched to allow another process, task or thread to reach the executing state. In this way processes take turns executing small pieces of code and parallelism is then said to be achieved.

Each time one processes is to be dispatched and another one preempted a **context switch** is called to perform the switching operation. Since processes can be dependant on each other concurrent execution brings the need for **synchronisation**. If more than one process require the same resource under simultaneous execution **mutex**, mutual exclusion, is needed to exclude all processes but one requesting that resource. Exclusion can prevent problems with data corruption or deadlock but it can also be used for synchronisation. A common way to perform mutual exclusion is through **semaphores**.

Continuing with typical real-time system terminology a second important field for RTOS discussions will be covered. The concepts are often used to determine the quality of multi-processing real-time systems and RTOSs.

- **Determinism**
Determinism in a real-time system means that the system is predictable upon time.
One piece of code must always take the same amount of time to execute in a deterministic system in oder to ensure that the deadline is always met.

- **Latency**
Latency is the time spent, or delay caused when, executing a section of code from beginning to end.

- **Response Time**
Response time is the time it takes for the system to react as an event occurs.

- **Throughput**
The throughput of a system is the speed at which an input value travels through the system to the output. In other words completed operations per unit of time.

These explained terms will now serve as a base for the discussions on parallel programming issues and RTOS design later in this chapter. Hopefully looking back at this section will facilitate while reading the brief description on basic RTOS design. For further explanations please refer to litterature on these subjects, [6],[2],[5],[3].

2.2 Issues of Parallel Programming

RTOS's are designed to facilitate resource scheduling in systems where several activities execute in parallel. There are many challenges in parallel programming as a number of race conditions can occur due to the concurrency. An RTOS has to include functionality preventing such conditions and therefore knowledge in parallel programming is required when developing RTOSs. Methods for avoiding the race conditions are crucial for operating system design. The most common race conditions will be discussed one by one briefly in this section, please refer to Ola Dahl [2] or other literature on parallel programming for further details.

- Deadlock

Deadlock is perhaps the most common and well known race condition as it is rather easy to achieve. Deadlock occurs when there exists a condition for execution that will never be satisfied, the system will lock waiting for that condition to be fulfilled. One example of when deadlock occurs is when two processes each hold a resource that the other process awaits.

Since both processes keep holding their resource until the requested one is set free they are both prevented from proceeding and the system locks [31]. Generally deadlock occurs when:

1. A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain. [38]
2. Each process waits for messages that has to be sent by the next process in the chain. [38]

Deadlock is defined as a ”Permanent blocking of a set of processes that either compete for system resources or communicates with each other”, professor Yair Amir [38]. Deadlock can only occur when a combination of system design choices have been made. This is when only one process at the time is allowed to utilize a resource and when the release of a resource is voluntary by the holding process [35]. This in combination with one of the above listed situations definitely cause deadlock. In order to avoid deadlock at least one of those conditions must not be satisfied.

Critical sections at risk of deadlock should be detected and handled carefully. For example one way would be to allow simultaneous access of a resource, another not to allow resource requests [38]. Simultaneous access could cause another race condition in its turn called preempted access which will be discussed next. In order to always avoid deadlock the RTOS should include a deadlock avoidance algorithm to render all unsafe states or critical regions with appropriate methods [35]. Read more about deadlock handling in the corresponding lecture by Yair Amir [38] and about shared resources in ”Real-time programming” by Ola Dahl [2].

- Preempted Access

Preempted access race conditions occur as two parallel processes are concurrently utilizing the same memory resource. If more than one process writes to the same memory the stored information could become corrupted or the information at that memory position would at least be misleading. This preempted access race condition forces the need of mutual exclusion to exclude other processes at such critical sections. That way only one process at a time will be granted access to a common resource. Mutual exclusion does not necessarily mean that the process can not still be preempted, only processes requesting the particular resource are excluded (relative mutex). Otherwise if the mutex does not allow any preemption at all it is called an absolute mutex [2]. Refer to Ola Dahl for more information on how to achieve mutexes [2].

- Synchronization and Communication Errors

Synchronization is needed forcing processes to wait at critical sections. As mentioned earlier such situations occur when several processes require shared data or when a process should wait in order to receive data from another process on which it depends. Waiting is also needed at communication between processes like when a process should wait for a specific event to occur or a message to be sent [2]. Synchronisation can be implemented using ”meeting points” in the program code, by event variables or by mutual exclusion.

- Priority Inversion

Priority inversion could occur if a process acquires a resource and is later preempted without first releasing that resource. The preempting process always has a higher priority and this could cause a problem if the process is requesting that same resource. If the low priority process does not release the resource the higher priority process will be blocked by the lower priority process waiting for that resource. In that way priority is inverted.

Certainly not all race conditions are mentioned here, only the absolutely fundamental ones. More about race conditions and examples of such can be read at Tom Sheppard's surreal-time page [31] or any literature on parallel programming. Avoiding all race conditions is the true challenge for an RTOS designer. Many different resource scheduling algorithms has been developed to optimize the RTOS functionality with respect to these conditions. In the following section you will learn more about the structure of a real-time operating system and how these challenges are usually faced.

2.3 RTOS Design

Real-time operating systems are widely used in embedded real-time systems as there are several obvious advantages with such an implementation. The RTOS does provide scheduling algorithms that guarantee deterministic behaviour of the system and it also provides a hardware abstraction layer. The performance requirements are particularly challenging in an RTOS in comparison with a commercial operating system due to the support for real time requirements. As defined in section 1.6 a real-time system is "a system that responds in a timely, predictable way to unpredictable external stimuli arrivals". Consequently "An RTOS provides facilities which, if used properly, guarantee deadlines can be met generally (soft real-time) or deterministically (hard real-time)" [29].

This abstraction is important for two reasons, simplicity and compatibility [32]. It simplifies the application development since applications does not need to be hardware dependant. The RTOS is usually designed for and compatible with many different CPUs and target boards which would then make your application compatible with may different system configurations. Using an RTOS also often facilitates debugging and scaling as many RTOSs provides the possibility to load an unload modules to the system dynamically [36].

The disadvantage of using an RTOS compared to using the foreground/background technique is the overhead particularly due to memory protection [36]. In simpler systems is it still preferred to implement multitasking without an RTOS to avoid unnecessary overhead. According to Lothar Thiele from the Swiss Federal Institute of Technology [37] there are three key requirements on an RTOS:

- Predictable timing behaviour.
- Time and scheduling management.
- Speed, the RTOS must be fast.

Obviously time and scheduling management is of great importance as it is the main purpose of the RTOS. The last requirement, speed, is said to be particularly important but it does not necessarily refer to a high system throughput. Instead the response time to events must be very quick and the interrupt and context switching latency should be minimal [31].

The RTOS is measured by its ability to predict execution times and be aware of task deadlines.

It should provide high resolution time services. The time it takes for the RTOS to react on an event and perform the necessary operations, must be predictable with a maximum time deviation. If the prediction does not match the truth the RTOS might schedule the resources in way that prevents processes from meeting their deadlines and cause the system to fail. All system calls in an RTOS must be deterministic, both operations and interrupts [33]. A good RTOS is reliable and thereby deterministic under all system load scenarios [36].

However, both memory management and I/O management can generally not be truly deterministic. Communication with peripherals and memory is unpredictable upon time [1]. Memory allocation time for example depends both on the size of the memory block to be allocated and on the fragmentation state [33] of the memory resource. The RTOS memory management unit (MMU) for memory protection should therefore include methods for avoiding fragmentation and unnecessary memory access. This unpredictability is handled by time bounded operations giving a guaranteed maximum time for the allocation.

An RTOS must always be "an operating system with the necessary features to support a Real-Time System" [36] and therefore the fundamental structure is basically the same for all RTOSs. According to Whittlesey-Harris [36] the RTOS has four major tasks being process management, interprocess communication (IPC), memory management and input/output management. The purpose of these task blocks is to provide the resource allocation, the hardware abstraction layer, an I/O interface, the time control and the error handling [33]. Figure 2.1 depicts the most common RTOS structure.

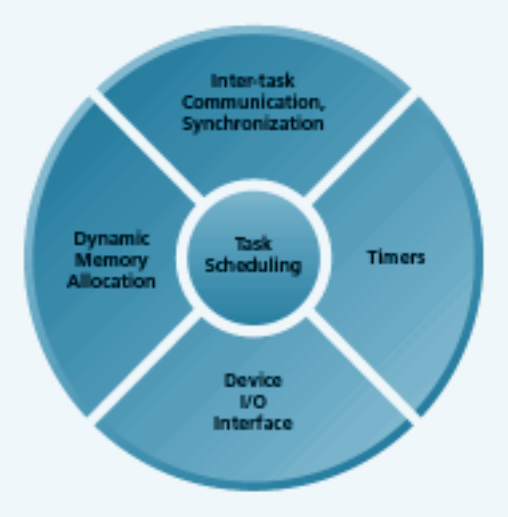

Figure 2.1: Basic contents of an RTOS [33], [44].

The task and resource management, meaning the scheduling of CPU and memory resources [36], is the central task of the RTOS. Information on how it is performed will be explained later in section 2.4. The I/O management block determines the hardware support and the memory block provides memory protection facilities through the MMU. The timer block controls system clocks and time resolution. Finally the interprocess communication and synchronisation block contains facilities such as mutexes, event variables and semaphores.

Interprocess communication is performed through mailboxes and message queues. Messages are sent to and read from the mailbox. The sending process is blocked if the box is full and the reading process is blocked until it receives the message that it awaits. When using message queues several messages can be sent before the sending process is blocked. The messages are placed in a first in first out (FIFO) buffered message channel. Examples illustrating these

[1] Memory allocation from a pool using alloc() is deterministic in OSE, using malloc() allocating memory from the heap on the other hand is highly indeterministic [7]

communication and synchronisation methods can be found in the slides presented by Ramon Serna Oliver [33].

The heart of the RTOS is the kernel, it is responsible for the hardware abstraction and the resource management [32] as well as process synchronization [31]. Applications can request services from the kernel through system calls using the RTOS specific shell commands for interaction with the hardware. Usually the kernel is small and highly optimized [33] including libraries for different target hardware configurations. Most RTOS design architectural decisions regard the role of the kernel [32]. There are several different kernel types deciding what functionality to be included in the actual RTOS kernel and what to be implemented as an external service. Figure 2.2 shows the RTOS layered structure.

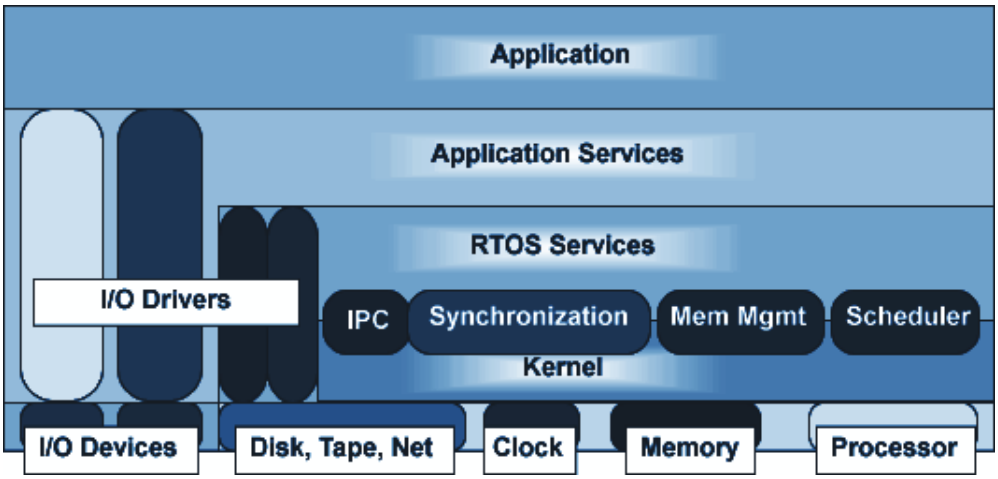

Figure 2.2: Common RTOS structure as depicted by Tom Sheppard [31].

As seen from the picture the RTOS kernel is connected to hardware peripherals and provides RTOS services. The blocks between the kernel and the RTOS services layer could either be included in the kernel or be implemented as a service. Which depends on the kernel structure. The application services layer is where you utilize the RTOS services customizing the system for your application. It serves as the base of functionality to be used in your application. Both RTOS services and the application services can access hardware directly without going through the kernel. Accessing hardware from the applications layer is desired if the RTOS does not include support for some particular hardware needed in your system. In that case you will have to write your own I/O driver to reach that hardware. Finally the applications layer utilizes the application services and hardware abstractions in applications [31].

With the RTOS structure in mind let us continue with further details on the actual scheduling techniques. The following section will explain basic strategies on how to provide good resource management.

2.4 Scheduling

Scheduling of tasks, processes and threads, in a system requires a scheduling policy for the RTOS to decide which process to run at a certain time. First consider the different kinds of processes that is likely to be present in the RTOS. Process characteristics and timing constraints decides how the process will be handled by the RTOS. Usually processes are divided after characteristics into different groups, here those groups are listed as by Sheppard [31].

- **Periodic** clock driven processes characterized by important deadlines.
- **Aperiodic** or sporadic event driven processes characterized by important response time.
- **Critical** processes characterised by strict response time maximum or CPU time minimum. Critical processes can be either periodic or aperiodic.
- **Idle** process running when there is nothing else to be done.

These different kinds of processes are not optimal to handle in the same way. Processes with different characteristics desire different scheduling policies. The most common scheduling policy practised is preemptive priority based scheduling [2]. Processes are assigned priorities and the process of current highest priority is basically always the one running. A context switch is performed as a process of higher priority than the currently running becomes ready. Remember the context switch and process states now depicted in figure 2.3.

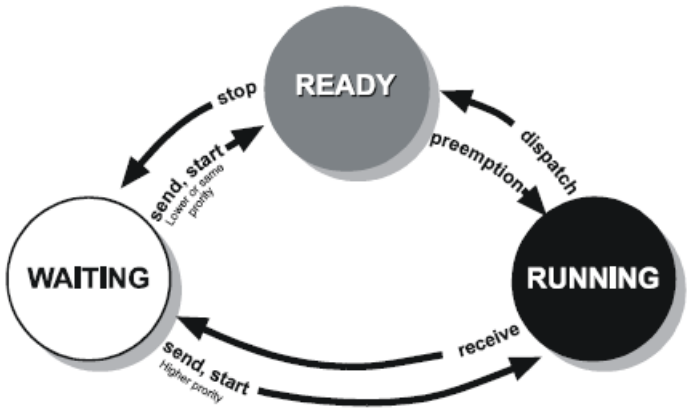

Figure 2.3: Process states and context switching [7].

To perform context switching all register values in registers used by the process to swap out has to be stored and remembered for the next swap in of that process [5]. Values are saved in a process control block, sometimes called a switch frame. As the process is swapped back in values are loaded from this process control block. The time it takes to save register values and load values for the preempted process is inefficient time overhead delay as no process execution can be performed concurrently. Therefore swapping frequency can have a large impact on the system performance. There is usually a queue of ready processes waiting that the scheduler needs to handle but if switching processes to often the system would suffer severe slow down.

Preemptive priority scheduling can either be of fixed or dynamic priority type. At "Fixed Priority Scheduling" (FPS) the process priorities are precomputed and static. Usually priorities are assigned to processes through an assignment scheme called the rate monotonic priority assignment scheme [3]. Dynamic priority scheduling enables priority changes during run-time. In a system using priority scheduling processes are also divided into the following groups.

- Prioritised processes
- Background processes
- Interrupt processes
- Timing Interrupt processes

Background processes are considered to be of lowest priority and interrupts usually preempt any prioritized process. System management processes are most often of highest priority. Prioritised and background processes can either be static or dynamic. A static process never terminates and is created at system start. A dynamic process is the opposite and can be created and killed as the system is running.

As mentioned earlier one scheduling scheme is rarely suitable for all processes in a system. Priority scheduling can be extended with a number of additional schemes. There are several operating system optimization objectives, some listed by Irv Englander [6]. Throughput should be maximized, response time should be minimized and be consistent, starvation should be prevented and the resource allocation should be maximized [6]. To achieve this cooperation between different scheduling schemes are necessary. To allow preemption between processes of the same priority the "Round Robin Scheduling" scheme can be added to cooperate with the priority scheme. Instead of priorities the round robin scheduling scheme uses time slices for CPU distribution between processes.

Another scheduling method is deadline scheduling, such as "Earliest Deadline First" (EDF) or "Least Slack Time First" (LSTF). As suggested by the name, at earliest deadline first the execution order is determined by the process deadlines. For periodic processes cyclic scheduling is desired. When choosing a scheduling scheme one should consider the process characteristics in the system. Combinations of different scheduling schemes is possible but complex.

2.5 Execution time prediction and WCET

Process execution time predictions are of great importance in an RTOS. As mentioned in the previous section on scheduling misleading predictions may cause the system to fail. The execution time is dependant on both hardware, software and the interaction between them. In modern embedded systems utilizing both cache memories and pipelines these execution time predictions are getting more and more difficult to perform correctly. Beside the preemption cost of context switching latency, caches and pipelines introduce severe problems as the microarchitecture of the system changes at preemptions [1]. Cache entries could become

displaced and instruction streams in a pipeline could become disrupted for example [1]. Of course these preemption costs have an effect on the execution time, in many cases calculating this time is infeasible. Safe upper bounds on execution time is essential to the scheduler using the worst case execution time (WCET) to guarantee deadlines. Caches and pipelines certainly improve the average performance increasing the throughput but at the same time they clearly increase the worst case execution time [1]. This is why in many cases the worst case execution time prediction is very pessimistic causing processes to receive more resources than actually required.

As new modern techniques arrive improving system performance new challenges are consequently brought to RTOS design. At the moment research is performed to better predict execution times in modern systems including caches and pipelines. Other approaches redistributing the otherwise possibly wasted resources are also currently of interest. Such research project is the FRESCOR project of which chapter 4 is about.

Chapter 3: OSE

OSE or "Operating System Embedded" is a real-time operating system developed at ENEA Embedded Technology AB, initially for the telecommunications industry [19]. It has been designed to meet requirements of the next generation complex embedded systems. Design focus has been reliability, scalability and simplicity. OSE is particularly suitable for distributed systems, especially telecommunications and wireless products. It is already used in millions of cellular telephones worldwide [19].

"The OSE RTOS makes it possible to quickly and easily create applications that operate at a higher level of abstraction and execute reliably over their lifetime" in contradiction to a conventional RTOS [19]. OSE supports dynamic software reconfiguration and has automatic built-in error handling. The heart of the architecture is the message passing mechanism that can easily be extended across CPU boundaries[19].

OSE promotes a specific programming model that enables modular design and easy debugging [46]. The systems developed with OSE will be of fault tolerant robust architecture [46] and the OSE board support package (BSP) support the latest standard target boards [19]. This chapter is a short summary and introduction to the OSE architecture.

3.1 OSE Architechture

OSE is an operating system optimized for message passing [7]. Message passing is particularly useful in distributed systems where communication between many different microcontrollers is needed [5]. The primary interprocess communication mechanism in OSE is asynchronous direct message passing [7]. Direct message passing means that no mailbox is used [5], communication is direct between two processes. Asynchronous communication means that non-blocking send and receive calls are used [5].

The communication interfaces are signals, semaphores, fast semaphores and mutexes in OSE. Signals are the simplest and most versatile method for communication [7]. They can contain data and are easily defined by creating a structure or class. Signals are very efficient and work well between processes on one processor as well as between processes on different processors. Signals do form a queue in a FIFO channel but one advantage with OSE is that it is possible to choose a signal at any position in that queue.

OSE is designed for simplicity in application development, the send and receive signal calls are two out of eight calls that is considered sufficient for most applications. OSE is built on microkernel structure allowing services to be implemented as plug-ins instead of being part of the kernel. An OSE process corresponds to either a process, thread or task in other operating systems and these processes are the most fundamental building block in OSE [7].

The OSE scheduler implements preempted priority based scheduling. Preempted fixed-priority scheduling is combined with round-robin scheduling and periodic scheduling for handling of different kinds of tasks [7]. OSE allows grouping of processes into blocks to improve the

system structure. Each block may have its own memory pool and consequently if one pool becomes corrupted only processes connected to that particular pool will be affected. The memory management unit (MMU) isolate different domains or segments from each other. This block architecture is illustrated in figure 3.1. There is one system domain containing the kernel and several application domains containing a memory pool for the application and a heap for dynamic memory allocation. In an OSE system there is always a system memory pool for global allocations [7].

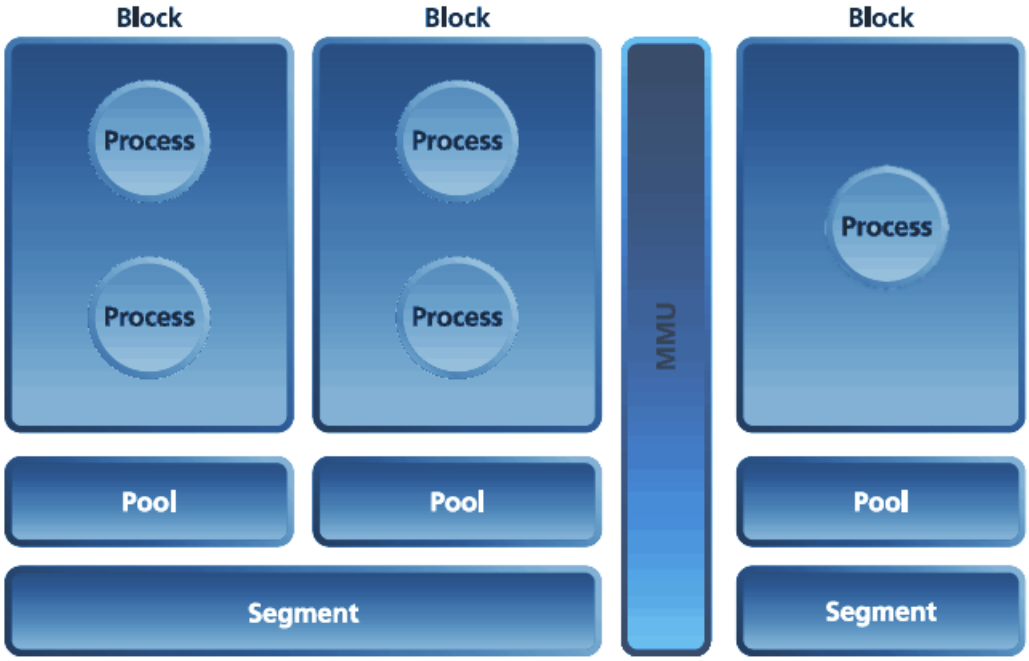

Figure 3.1: OSE memory configuration example [45].

One unique OSE feature is the error handler. There are four levels predefined for error handling, the processes level, the block level, the system level and the kernel level. As an error is detected within one of these contexts the corresponding error handler is called. The task of the error handler is to either fix or acknowledge the error and return to the caller. A second alternative is for the process error handler to terminate the calling process, the block error handler to terminate a block or likewise. If the called error handler can not solve it in either way the next level error handler is called for a try [7]. This error handling functionality and the architecture of loosely coupled memory domains provide OSE with the facilities to be classified as a fault tolerant system.

Chapter 4: FRESCOR

A majority of RTOSs today are designed to meet the demands of a hard real-time safety critical system. However, many areas of application as multimedia systems for example do often have more soft real-time requirements. A system with both hard and soft real-time timing constraints requires flexibility to handle all the different types of processes and their requirements [16]. In such systems there are often requirements for flexible sharing of resources and it is also not unusual that requirements change dynamically [16]. A scheduling framework for such a system of both hard and soft real-time applications is currently being developed in the FRESCOR project, namely a *"Framework for Real-time Embedded Systems based on COntracts"*.

A trade-off between predictability and efficient resource utilization is realised in FRESCOR through possible selection of the desired scheduling scheme. Cooperating scheduling schemes will handle each task after how it should optimally be treated by the system [16]. Between applications and the scheduler is an interface called the service contract. Through this interface applications negotiate for services. The framework means not only to schedule the processor and memory but also the network utilization and the disc and bus usage [16].

ENEA is a participant in the FRESCOR project with the aim of porting this framework to the OSE real-time operating system. This chapter will explain more about FRESCOR and the tasks to be performed at ENEA.

4.1 FRESCOR Background

FRESCOR is a research project in part funded by the European Union. There are six participating universities and five companies of which ENEA is one. ENEA participates contributing as an RTOS developer [25] to test the scheduler on a non-posix RTOS [18]. Work on the FRESCOR scheduler is based on a prototype of a flexible scheduling framework developed in a project under the name FIRST [16]. The aim of the FIRST project was primarily to support:

- "Co- operation and coexistence of standard real-time scheduling schemes, time- triggered and event-triggered, dynamic and fixed priority based, as well as off-line based" — [26]
- "Integration of different task types such as hard and soft, or more flexible notions, e.g., from control or quality-of-service demands, and fault-tolerance mechanisms" — [26]

The FRESCOR project will extend the aim of the FIRST project to further improve flexibility through contact based scheduling. Read more about the requirements on the FRESCOR scheduler in the architecture guide [16]. Research and development at the six universities has resulted in the FRESCOR framework, the task of the companies on the project is to evaluate and exploit this framework. The specific task for ENEA is to implement the FRESCOR framework on the OSE operating system. Other companies will be developing a simulation environment, applications and analysis tools for the scheduler [25].

4.2 Scheduling Based on Contracts

Contract based design is the key feature of the FRESCOR framework to further optimize resource utilization. An application negotiates a set of contracts with the scheduler specifying the application requirements. Negotiation occur at initialisation or dynamically as requirements change or when new software is added [16]. As a contract is accepted the system guarantees at minimum the requested resources and if available desired additional spare capacity can be granted.

Sometimes the WCET predictions can be rather pessimistic, as mentioned in section 2.5, due to complex system structures with caches and pipelines. Some processes might reserve more resources than actually needed while other processes would benefit from more processing power than requested. The contract based scheduling technique has been developed to better distribute these resources as it is discovered that a prediction was pessimistic by renegotiation of contracts during runtime.

A virtual resource is created as a contract is accepted representing the reserved resources. It stores information on the reserved resources and registers how much that has already been consumed by the process. This resource consumption and elapsed execution time indicated whether redistribution of spare capacity is appropriate. Virtual resources is a key feature when it comes to dynamic reconfiguration of requirements, because of it renegotiation might not be necessary as requirements change. "No contracts need to be renegotiated when the association between thread and virtual resource is changed since the virtual resources are separate from the actual entities being scheduled" [16].

Application contracts are divided in groups by the scheduler after their implementation complexity. Some contracts may contain several challenging requirements and others only few. The contract complexity is determined by the application requirements specified. This enables a modular implementation of the FRESCOR framework where different modules handle different contract requirements. These modules are illustrated in figure 4.1. The core module contains operations required to create and negotiate contracts, it also contains contract information related to the application minimum resource requirements. The information could be type of contracts, resource type, deadline, minimum execution budget and so on. [16]

An other module handles the shared resources and contains a list of critical sections, the spare capacity module contains attributes for distributing any left-over capacity and likewise there are modules for memory management, energy management and so on. The FRESCOR framework API FRSH, pronounced "fresh", provides services from each module.

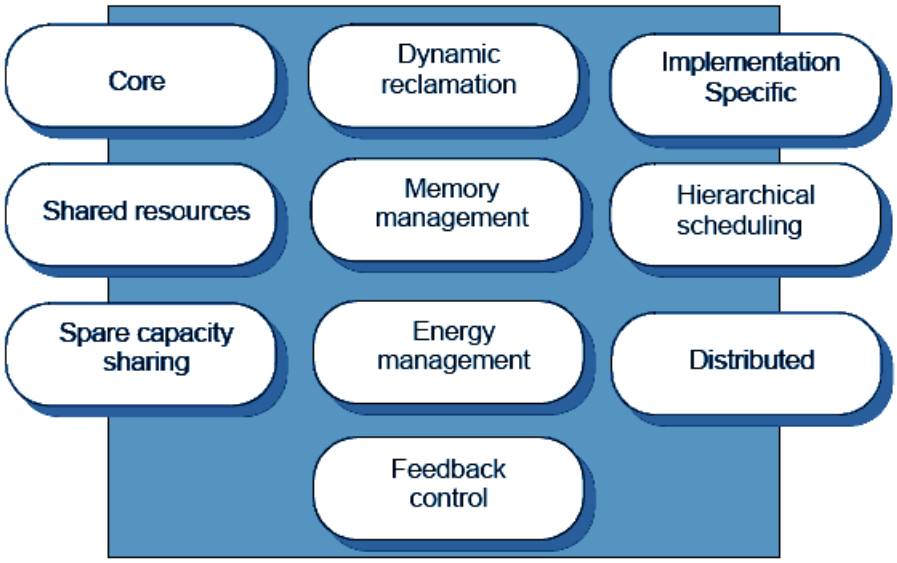

Figure 4.1: Modules in the FRESCOR framework [16].

4.3 FOSA on OSE

The FRESCOR framework FRSH is designed to be implementable on any RTOS and should not be operating system dependant. Therefore an adaption layer to each operating system to use FRSH is needed. FOSA stands for "FRSH Operating System Adaption layer" and should provide the minimum required functionality needed by FRSH.

FOSA for OSE is what is currently being developed at ENEA for the FRESCOR project. It will translate FRSH calls from the scheduler to ordinary OSE calls. The development of FOSA for OSE is lead by Erik Thorin at ENEA Services Linköping.

Chapter 5: Execution Time Measurement

Two important parameters are to be considered for accurate process execution time measurements. First of all, measurements must be performed at the exact time of a context switch. Secondly, time has to be measured at the desired time resolution. Finding an appropriate method producing a minimal measurement latency and sufficient time resolution is crucial for the solution to be feasible.

In case these requirements are not fulfilled misleading information will be delivered to the scheduler. That could cause adverse scheduling decisions and consequently processes will either miss their deadlines or resources will not be correctly utilized. In this chapter the available OSE services for time measurement will be discussed and their respective time resolution will be evaluated.

Found methods that could possibly realize process execution time measurements will be explained. Measurement accuracy discussed and resolution will be considered for each method respectively. Later in chapter 6 these methods will be evaluated in the purpose of determining the most appropriate method for implementation.

5.1 Measuring Time in OSE

The three system calls below constitute the built-in OSE timing functionality. Time is usually represented by a number of system ticks that is incremented after a certain time interval. The duration of a tick is different for different system hardware configurations.

- get_ticks()
- system_tick()
- get_systime()

The first system call, get_ticks(), returns the number of system clock ticks since system start. Using this call for measuring time is only an alternative if time at the tick resolution can be considered sufficient. In the OSE soft core environment one tick is 10,000 micro seconds and we will later see that at in the environment chosen for hardware implementation the tick length is 4,000 micro seconds. The resolution requirements for the FRESCOR project are not yet known. However it is unlikely that tick resolution is sufficient since context switching will most likely occur several times in on system tick. The system_tick() call can be used to return the system tick length in the current system. It is used when converting a time represented by a number of ticks to a number of microseconds.

The most realistic choice for time measurement is to use the get_systime() call. It returns the number of elapsed system ticks since system start and the number of microseconds that has past since that last tick. The resolution will consequently be at microsecond level which is much more likely to be sufficient. The microsecond counter does unfortunately only rarely

provide accurate timing reference, the resolution presented by the get_systime() call is hardware dependant. Read more about this restriction in the OSE system programming interface reference manual [13].

There is another alternative for even better resolution utilizing a hardware timer. That would definitely accomplish sufficient time resolution. There is a hardware timer interface in every board support package (bsp) layer. The disadvantage is of course that the solution will be hardware dependant. The timer implementation will then have to be made for a number of system hardware configurations in order to ensure compatibility with all hardware configurations that might be desired. That would be all target boards that OSE is compatible with.

The execution time measurement is necessary to support correct use of the FRESCOR scheduler. Therefore a method always available and accurate is essential to provide the necessary time stamps. Allowing hardware to restrain compatibility should therefore if possible be avoided. The conclusion is consequently that the *get_systime()* call is most suitable for time measurements if the microsecond counter works as intended.

5.2 The RMM Implementation Method

The Run Mode Monitor (RMM) is a module mainly used for debugging and load profiling. It constitutes a debug support interface that can give information about most system objects during run time [9]. It is possible to use the RMM for execution time measurement through its signal interface.

The run mode module is new for the OSE5 distribution and replaces the previous debug server core extensions component. It is possible to implement a context switch notification through using the predefined RMM signal called MONITOR_SWAP_NOTIFY. The notification signal contents is a struct called MonitorSwapCreateKillInfo. It includes information on which process was swapped in, which was swapped out and at what time [13] it occurred. The time stamp has a resolution at microsecond level presented in system clock ticks and microseconds since the last tick. This representation, in similarity with the *get_systime()* results, is sufficient resolution for this purpose.

Receiving a notification at each swap and identifying the processes enables process execution time measurement. Saving the time stamp for a processes at swap-in will create the possibility to calculate the time of an execution slice at swap-out through simple subtraction. Adding the slice time to the time of any previous slices a total execution time for that process is held. Choose a suitable container for storage of the results and the work is done.

The run mode module or monitor is implemented as a process. The RMM module will become enabled when it receives a signal of the type *"MonitorConnectRequest"*. In order to send that signal to the monitor the process id of the RMM is needed. A *hunt()* call for the processes by the name "ose_monitor" returns this process id. The id is then used in the *send()* call when sending the particular signal that requests a connection. If the measurement application has specified the MONITOR_SWAP_NOTIFY in its signal selection the *receive()* call will await notification. View the code example for connection to the RMM in figure 5.1.

```
/*Signal Selection*/
union SIGNAL *signal, *monitor_reply;

static const SIGSELECT selectMonitorReply[] = {2,
   MONITOR_CONNECT_REPLY
   MONITOR_SWAP_NOTIFY
}

hunt(''ose_monitor'', 0, &rmm_pid, 0);

/*If RMM found*/
signal = alloc(sizeof(MONITOR_CONNECT_REQUEST), MONITOR_CONNECT_REQUEST);
send(signal, rmm_pid);

monitor_reply = receive(selectMonitorReply);
```

Figure 5.1: Connect to the RMM.

If the reply from the monitor is a granted similarly request swap notification. The signal interface in OSE is known to be a very fast form of communication. Even so it is not necessarily enough for this execution time measurement application. Context switching is extremely frequent and signals could form a queue if data is not processed faster than it is sent. The time measurements performed by the RMM are accurate but there is no way of guaranteeing that the latest measurement results are actually available when the scheduler makes a request for it. According Magnus Karlsson [42] at the OSE core development team, ENEA Stockholm, "The RMM is not an alternative if measurements are performed a hundred times in one second, but once per second is probably okay and in that case no queues should arise". Measuring once per second, meaning once per millionth microsecond, is definitely to rare. As mentioned before context switching could occur several times in one system tick.

5.3 The Kernel Handler Implementation Method

Additional operations to be performed at context switching can be added as an extension in OSE through so called kernel handlers. There is a create handler that would be called at process creation, swap handlers called at context switching and a kill handler. Consequently as a process is swapped in or out extra operations to be performed can be invoked by the user.

Activating these swap handlers is one possible method of measuring execution time. By measuring the time once in the swap in handler, once in the swap out handler and then calculating the time difference, elapsed execution time for one slice can be determined. Adding all slice execution times for a process together gives the total elapsed execution time for that process. The measurement is guaranteed to be performed at the exact time of a context switch and will therefore be accurate.

The disadvantage using this method for execution time measurement is the increased context switching latency. Unless the extra operations are performed very fast an unacceptable system overhead could be the consequence. There is therefore a limitation concerning allowed system calls from inside a kernel handler. The OSE core user's guide manual [8] even declares that no system calls could be made from kernel handlers. However, this is not completely true. Which system calls that are available depends on the specific system and which calls that are unsuitable depends on the system requirements.

For execution time measurement the *get_systime()* system call to retrieve time is necessary. Memory allocation through the *heap_alloc_shared()* call for storage of the measurement results must also be possible. Both these calls are possible and consequently the kernel handler implementation is an alternative solution for execution time measurement. This is if the used system calls does not turn out to be unsuitable and if unacceptable overhead can be avoided.

The speed requirement brings challenges to the implementation. Particularly concerning storage of the measurement results, access to the correct container position must be fast. Assume that to each process belongs a node in a linked list created to store the results. To identify the list node specific for a certain process iteration through the list is possible. However, iteration through the container can not be allowed since it might cause a large time overhead.

This problem can partly be solved by utilization of the "user area" core component extending process specific memory. This memory is directly accessible from the kernel handlers however, not outside them. This memory can be used to store a pointer to the container position for fast access. Still one iteration has to be performed to find the position the first time.

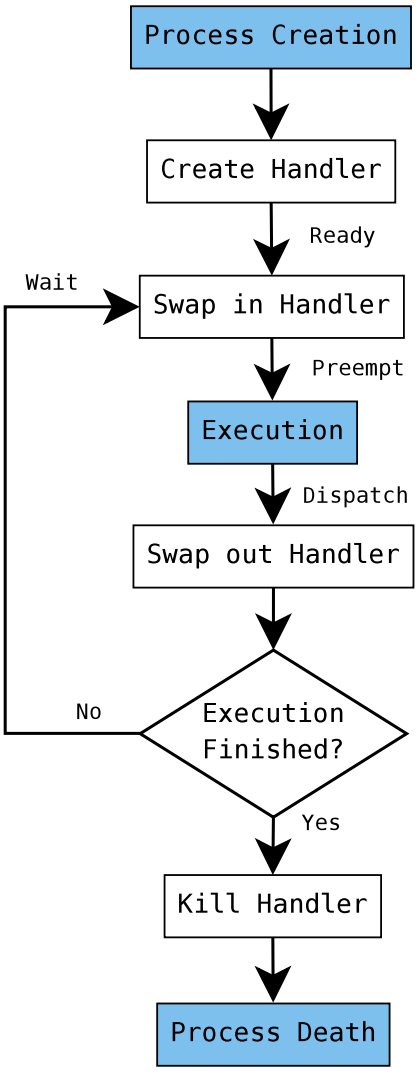

Figure 5.2: Kernel handler implementation flow chart.

5.4 OSE Kernel Modification

The execution time measurement functionality can be included as a kernel service. This would probably be the absolute most accurate and possibly fastest way to perform the measurements. The kernel implementation of a context switch could be updated to include the extra operations fore execution time measurement.

Such an implementation does not differ significantly from using kernel handlers. A kernel modification would only move the code otherwise placed in kernel handlers to the kernel directly. There would not be any difference in accuracy or resolution between these two alternatives. Whether a kernel handler or a kernel modification implementation becomes satisfying depends mostly on the code quality. It is however desired that only experienced programmers are allowed modifying the kernel.

Modifying the OSE kernel also requires the development of a new OSE kernel distribution. Another disadvantage when including the functionality inside the kernel is less flexibility. If for some reason you would not like to fully utilize the FRESCOR scheduler, perhaps several scheduling alternatives are available, the choice of using execution time measurements or not might be desired. Still, if the FRESCOR scheduling method is to be used so is the execution time measurement.

Chapter 6: Measurement Method Evaluation

Three alternative methods for execution time measurement has been discussed in chapter 5. These alternatives will now be considered for implementation. This chapter will explain parameters needed to evaluate the solutions and determine the one most suitable for implementation.

Beside the accuracy and time resolution parameters for evaluation this discussion will also include overhead and flexibility. The pros and cons of each measurement method will be declared and compiled into a table. That table will serve as a foundation for making the final implementation choice.

6.1 Analysis Approach

In order to analyze the feasible solutions of execution time measurement, parameters determining the most preferable solution for implementation must be declared. Below is a list of such parameters.

- Accuracy
- Time resolution
- Context switching latency
- Flexibility
- Memory Overhead
- Complexity

Accuracy is a parameter describing how exact the measurement results are, telling whether the measurements take place at the exact time of a context switch or not. If there is delay between actual swap and the time stamp received the accuracy is affected. This parameter also considers whether the result will be instantly accessible to the scheduler or if there is delay before the results are handled.

Time Resolution as mentioned in chapter 5, represents the unit in which time is measured. For instance time measured in microseconds has better and more sufficient time resolution than if measured in system ticks only. Time resolution also affects the measurement accuracy.

Context switching latency is the parameter telling how much extra time per swap that is required to perform the measurements. Dynamic scheduling, optimizing resource allocation in order to improve the system performance, would loose its purpose if the execution time measurement causes a severe system slow down due to high context switch latency.

Flexibility is a parameter for determining the hardware dependency. A flexible solution can be used on different hardware configurations and processors.

Memory Overhead is caused by unnecessary or extensive memory utilization. Memory is a limited resource and should therefore be spent carefully.

Complexity is the parameter for considering implementation time and difficulty as well as development cost.

6.2 Evaluation and Choice

The memory overhead can be expected to be very similar in the three cases as the same temporary values and non-temporary results need to be stored regardless which method is used. Considering complexity only the kernel modification methods stand out. That leaves four parameters particularly important to investigate for consideration in table 6.1. The characteristics of each method is graded with one, two or three stars. One star indicates that the method does not behave as desired and three stars indicated good behaviour. A question mark indicates a dependency or a condition that could change the grading.

Method	Accuracy	Resolution	Latency	Flexibilaty
RMM Implementation	*	**	**?	**
Kernel Handlers	***	**	**?	***
Kernel Modification	***	***	**?	*

Table 6.1: Method evaluation table.

Development of a new OSE distribution is not preferable at this development stage particularly since the method is not flexible. This is why, if one of other two methods are appropriate for implementation, the kernel modification method will be ruled out. Using swap handlers a resolution on microsecond level can be achieved as in the case with the RMM. This resolution is not as good as when using hardware timers but still good enough. Using the RMM requires an implementation to be on OSE5 which is why that method is considered slightly less flexible than then the kernel handler implementation.

The extra context switching latency caused at an RMM implementation is not known exactly but according to Magnus Karlsson [42] the time overhead is not to bad. The context switching latency of the kernel handler and kernel modification implementation both depend on how well the programmer can speed optimize the code. This makes it difficult to draw any conclusion to which method is better in this aspect. Using the RMM however there is a risk that swap notification signals will arrive frequent enough to form an increasing queue. In that case there might be an unacceptable delay before the correct value is saved and that affects the accuracy in a negative way.

The time overhead in a system implementing any of these methods for execution time measurement depends on how many swaps that will be performed. Consider the quote from section 5.2 ”The RMM is not an alternative if measurements are performed a hundred times in one second, but once per second is probably okay and in that case no queues should arise”. That in combination with a second quote by Magnus Karlsson [42] ”The RMM will always be slower than the swap handlers but the overhead is not terribly bad, the suitability depends on the application” clearly explains that the choice between these two methods is not obvious.

Since at this point conclusions regarding the swapping frequency in possible applications can not be drawn a kernel handler implementation seems to be the most appropriate method. Particularly since there is most likely more than one swap per second. Chapter 7 will explain the implementation of the kernel handler method for execution time measurement.

Chapter 7: Implementation

As mentioned in chapter 5, an implementation of execution time measurements using swap handlers needs to be speed optimized. Overhead slowing down the system or an unnecessary resource utilization would not make this alternative solution a realistic approach to the problem. To the conclusions made in chapter 6 accordingly a swap handler implementation is the most appropriate alternative solution for realisation. Still we need to address the particular swap handler design issues mentioned in section 5.3.

This chapter will in detail describe the software implementation of execution time measurement through kernel handlers. Design issues will be considered and choices will be explained and a test application with the flowing results will be presented and analyzed.

A presentation on the available software tools of interest will be given in this chapter initially, together with an explanation on how to configure the environment. Secondly there will be a discussion on how to minimize measurement time and memory usage in a swap handler implementation. The memory configuration choices resulting from this discussion will be implemented and explained in section 7.4 The kernel handler implementation is presented in section 7.5 and thereafter the implementation of a test application.

Results from running the test application will be discussed in section 7.6 and conclusions concerning time and resources motivating later hardware testing will be declared. Finally after that some comments on the integration of this software with FOSA will be mentioned.

7.1 Tools and Environment Configuration

Initially, before starting any implementation, development tools for OSE and the source code programming will need some configuration. Assuming that you have the OSE5.2 distribution installed on a Windows computer this section will explain how the environment is configured. All settings to be made after the OSE installation for this project will be mentioned briefly. Some settings are very general and some are more application specific.

License settings must first of all be made in order to enable the OSE installation. Assuming you have a license file, to configure OSE with the license file in windows go to my computer and properties. Choose advanced and then environment variables. Create a user variable with the name LM_LICENCE_FILE and define it with the name of the license file. Then, add the path to your license file under system variables. Do so by marking the path line and choose edit. Do not remove any existing paths but simply place a semicolon for path separation. You can read more about license setting in the OSE Getting Started user's manual[10].

Cygwin is a program to be installed on your windows computer, if Cygwin is not already installed. Cygwin is required to provide a Unix-like development environment in windows needed in order to run the OSE reference system (refsys). Cygwin is open source software and can be downloaded from the Cygwin homepage [21] but it is also shipped with OSE. It can be found in the OSE5.2 catalogue, often referred to as the OSEROOT. The version of Cygwin included in the OSE installation has the advantage that it has been tested to work with OSE. In order to use Cygwin from a command window, add the path to the Cygwin/bin

directory in the same way as the license path was set earlier. Of course, these path settings for both license and Cygwin could also be made using the command prompt. See the OSE Getting Started User's manual [10] for further instructions on these configurations.

With Cygwin installed the OSE reference system can now be used. A reference system is a platform for custom OSE system design that simplifies the usage of the make system. There are two different reference systems to choose from in OSE providing different functionality in shape of command line shell available to the user. The two reference systems are called POLO and RTOSE. POLO is a bootloader reference system, Portable OS Loader, to be installed on a board flash. Obviously this is not available in the softcore environment. RTOSE, real-time OSE, is a highly configurable generic platform containing most OSE functionality. Real-Time OSE will be the reference system used in this project, read more about reference systems in OSE Getting Started [10].

Reference System settings for adapting the RTOSE reference system to the specific project are next in line. The reference system includes the concept of creating applications as modules. For your application you need to choose whether to design a load module or a core module. A load module is loaded to the reference system during run time while a core module is linked as a library to the OSE core and thereby included at the softcore build. The core module application is not needed by the reference system during start up and therefore it is not activated until late in the system start procedure.

The execution time measurements module will be linked with the OSE core as core module. This is since measurements need to be activated as soon as an interesting FOSA process is created and therefore the measuring function always needs to be available. In order to include the core module to the RTOSE build, configurations in the specific makefile for the OSE core to use with RTOSE need to be made. Go to the folder OSEROOT/refsys/rtose and choose the catalogue of your core. In this project the sfk-win32 softcore will be used and in that folder you can find a file called rtose.mk which is the specific makefile for the softcore. In that file add the line below to include you core module in the build

```
override MODS += modulename
```

This is the case when the module is saved among the other modules in the OSE/refsys/modules catalogue. It is also possible to add external modules, saved outside the OSE directory. That is done with the XMOD, instead of MODS, command. Read more about linking external modules in section 10.2 of the OSE Getting Started User's manual [10].

Build flavor and architecture settings are next in turn. Modules can be built in different flavors, debug or release. In this case the debug flavor build is the one of interest; it is an optimized flavor for error detection and debugging. The release flavor is instead optimized for performance and footprint. Settings for the flavor of choice as well as the processor architecture in your system needs to be made in the environment.mk file in the refsys catalogue. In this case the softcore will be run on a PC will be an Intel x86 processor. To set these architecture and flavor choices to be default settings, configure the environment.mk file by modifying the ARCH and FLAVOR lines to the following.

```
#Default target architecture and flavor for module builds
ARCH ?= x86
FLAVOR ?= debug
```

Now most of the environment configurations are set. The configuration file, rtose5.conf, is used when starting the soft core. In the Cygwin command window, go to the sfk-win32 catalogue. Type make all to compile the soft core together with the new application module, or remove the override MODS line in the rtose.mk file to compile without your module. If no compilation errors occur, start RTOSE by typing:

```
obj/rtose_debug/rtose -c rtose5.conf
```

An executable file, rtose.exe, has been created at compilation in the sfk-win32\obj and then rtose\debug catalogue. It is executed when running the above command according to the specifications in the configuration file, rtose5.conf. When the soft core is running you can find a list of which shell commands that are available by simply typing help. For instance you can list the processes in the system with the ps command. If you prefer a graphical interface you can find the same information in your web browser by typing the IP-address given at the RTOSE start up in the web address field. Now, the system should be running correctly.

7.2 Design Issues

Implementing the kernel handler alternative solution for execution time measurement requires considerations concerning the kernel handler specific design issues. Both memory utilization and the speed of the measurements are important parameters when designing an implementation realistic for future use. To much overhead is not acceptable when the purpose of the measurements is to improve the system performance. As mentioned in chapter 6 the kernel handlers are especially appropriate due to advantages considering accuracy in the measurements. This accuracy comes with a cost of context switching latency. How much latency and if that latency can be tolerated or not, is what will be investigated through testing this implementation.

Iterating through a container in order to save the measurement results at the right position will as mentioned cause a relatively large timing overhead. List iteration therefore needs to be avoided. The problem being the first design question 7.1 is how to, in the fastest possible way, locate the container storage position for each process. If the container memory is allocated at process creation inside the create handler a pointer to that position could immediately be stored in the process specific memory for fast access at every swap.

Design Question 7.1
How can the execution time result storage position for each process be located in the fastest possible way?

If the container memory allocation does not occur inside a kernel handler we must perform this iteration at least once. It is possible to overload the create process function and allocate the container memory in that function. In that case it will not be possible to identify the position inside the create handler since the process id can not be saved inside the container for comparison until after the create handler has already been invoked. As the *create_process()* is called inside the overloaded function the create handler is invoked, the process id is returned from the *create_process()* call meaning that the id can not be stored in the container before the create handler has been invoked. Consequently there is no process id in the container for comparison while inside the create handler and the container position can therefore not be identified inside the create handler if it is not allocated there.

One container iteration has to be made and if not in the create handler it has to be inside one of the swap handlers. The swap handlers are, unlike the create handler, invoked several times for one process. Therefore we need to find a way of avoiding repeated iterations through the container. For instance, controlling if this swap is the first swap for the current process.

Optimizing speed and memory does not always go hand in hand. In the first case when

allocating memory inside the create handler we will unfortunately allocate container memory for all processes in the system, interesting or not for execution time measurement. Avoiding global variables (desired for good project structure and readability of the code) there is no way to determine if the process is a FOSA process as the create handler is called, being once for every processes in the system. The most time efficient method obviously causes memory overhead. This is another design issue formulated in question 7.2 below.

Design Question 7.2
How large impact does the time of one list iteration per process have on the application performance? Is it worth performing one iteration to avoid memory overhead?

This obviously depends on the number of positions in the container to search, meaning that it depends on the number of processes in the system. If there are many processes unacceptably long iterations could be required. On the other hand, the positions to search is likely relatively few if:

- Memory for new positions is allocated at the beginning of the container.
- A search always starts at the beginning of the container.
- The iteration is performed only once per process at the first swap in.

Likely the first swap in occurs relatively soon after the process creation, then also soon after the memory allocation which is at the beginning of the list. Consequently the node will be positioned relatively early in the container at the time of the first swap in and only few nodes will have to be searched. Would this optimization affect the trade off choice?

This is a compromise between time and memory overhead but in a large system with many processes competing over the available memory both parameters are important. To minimize the waste of shared memory a dynamically linked list is chosen as a container for storing the measurement results. In that way a list node can simply be deleted when the information is no longer of interest and memory area is not permanently allocated or wasted. The compromise between time and memory overhead decreases the memory overhead as list nodes will only be created for processes of interest for execution time measurement.

Still, the process specific memory created for all processes will be needed for the process id comparison. Some memory overhead is therefore inevitable. The user area is suitable for storing temporary variables, such as slice start times, that are not of interest to any other process. The advantage of the user area is that it is automatically deleted at process termination. The global container on the other hand requires manual deletion of list nodes which could easily be forgotten and cause devastating memory overhead. The question of where to store the majority of variables is the second design question 7.3.

Design Question 7.3
Which causes the most memory overhead, extending the user area for all processes or storing information globally for FOSA processes?

Fully utilizing the user area memory is a safer way to ensure freeing not needed memory. The implementation presented in this chapter will constitute the compromise alternative with one list iteration, minimizing the number of list nodes to search. Hopefully this trade off will turn out to be a feasible solution. Which is really the most advantageous choice depends on the system application and the number of processes requesting memory.

7.3 Project Structure Settings

The source code implementation of this project will be developed in the C programming language according to FRESCOR semantics [14]. The C code will be written in eclipse, an open source editor, configured with OSE reference documentation. To make this configuration, add the OSE documentation plugin files to the Eclipse plugin directory. There are several OSE plugins for Eclipse that extends the Eclipse CDT plugin needed when writing C or C++ code. See the Eclipse for OSE User's Guide for more information [8].

Create a standard C project in Eclipse, this project is called fosa_ose_exe_time. In this folder there should be a source code folder containing the file fosa_ose_exe_time.c and header files that might be needed for the application. Beside the source code you should have a makefile, an application specific makefile fosa_ose_exe_time.mk and a main configuration file osemain.con.

Targets for the core module build is specified in the makefile and the environment specific makefile, environment.mk from the refsys folder, is included. Object files to be created from the C files are also specified in the makefile and the path to the source code is set, finally the modules.mk file is included for the module build. Note that the order of these specifications are important. In the fosa_ose_exe_time.mk file project specific build options are set, such as the path to the osemain.con file and the path to the library files to be created. The makefile can be viewed in appendix F and the projext specific fosa_ose_exe_time.mk file in appendix G

There are some configurations needed to invoke the kernel handlers and the user area respectively, see figure 7.1. These settings are made in the kernel configuration file, krn.con located in the core specific folder. An example kernel configuration file is available in appendix A.

```
/*   Activate kernel handlers for process execution time measurement    */
CREATE_HANDLER    (fosa_ose_create_handler)
SWAP_IN_HANDLER   (fosa_ose_swap_in_handler)
SWAP_OUT_HANDLER  (fosa_ose_swap_out_handler)
USER_AREA         (21) /*21 extra bytes of memory per process*/
```

Figure 7.1: Kernel handler and user area activation.

The functions implementing the kernel handlers will have the names specified within parenthesis. The user area for each process is extended with the number of bytes specified in this file. It is also possible to define your own shell commands in order to activate the application module processes. Such configurations should also be made in the krn.con file.

7.4 Memory Configuration

The user area process specific memory is extended by 21 bytes for each process as defined in the kernel configuration file. This size must correspond to the size of the implemented variables to be stored in the user area. For this application that would be process id, number of ticks, number of microseconds, a pointer to the process list node and a boolean variable telling whether the process is a FOSA process. The twenty one bytes is motivated as described by comments in the user area structure in figure 7.2.

```
typedef struct user_area{
    OSTICK tick_slice_start_time;               /*unsigned long = 4 bytes*/
    OSTICK micro_slice_start_time;              /*unsigned long = 4 bytes*/
    fosa_ose_list_node_t *list_position_ptr;      /*pointer size, 4 bytes*/
    OSBOOLEAN fosa_process;                 /*size of unsigned char,1 byte*/
    PROCESS pid;                                /*unsigned long = 4 bytes*/
}user_area_t;
```

Figure 7.2: User area configuration.

As seen in the code of figure 7.2 the variable sizes add up to exactly 21 bytes. The OSE specific type definitions can be found in the system programming interface manual [13]. The user area memory is used for the temporary slice start time variables used at calculation of the execution time while the result from that calculation will only be saved in the list node. The user area also contains the process id for node identification in the swap in handler and a pointer to save the address to that node when found. Finally the boolean variable is used to determine whether to perform the execution time measurement or not.

The list node structure should contain process id, total elapsed execution time described in ticks and micro seconds and pointers to the next and previous nodes. A header file is created containing the node structure and an initial empty header node to the list is predefined there. Also the delete node function is declared, but not defined, in the header file. This header file can be found in appendix D.

Only the necessary information required by users of the measurement function is globally stored in the list nodes. The user area will be initialized in the create handler and the list node will be initialized in the overloaded create process function. Variables are set to zero and pointers are set to NULL during initialization except the process id that is written to both memories as soon as possible.

7.5 Kernel Handlers

The discussion in section 7.2 considering design issues has determined the contents of each kernel handler respectively. Let the detailed explanation of the kernel handler implementation begin with the create handler. The create handler is activated once for each process during the process creation. Because of the time aspect it is preferable to keep the contents of the kernel handlers as short as possible, particularly in those called often. Therefore we would like to perform as many of the operations as possible inside the create handler instead of inside the swap handlers. As mentioned before the list iteration can unfortunately not be performed inside the create handler with the chosen memory configuration. However, we will store initial values in the user area and most importantly we will save the process id. A flow chart displaying the create handler functionality is depicted in figure 7.3.

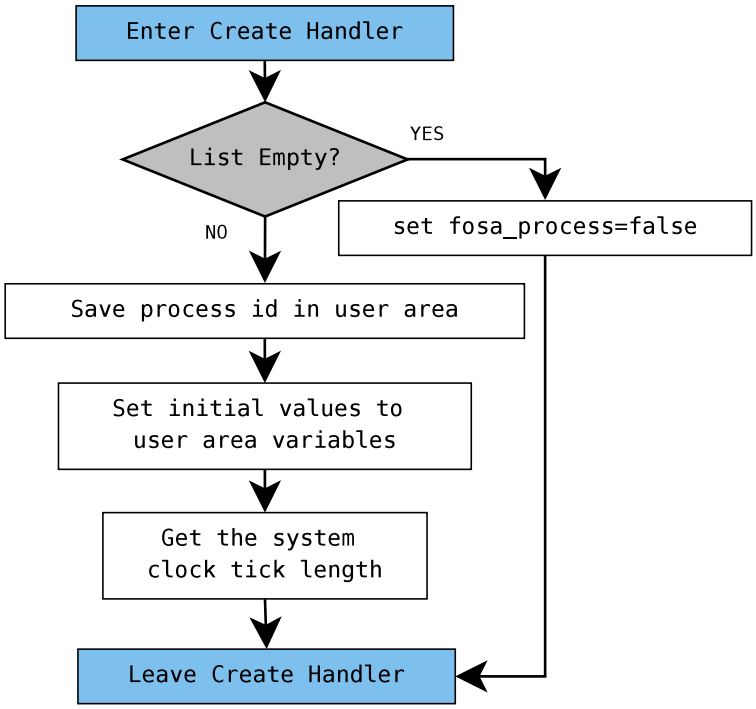

Figure 7.3: Create handler flow chart.

Remember from section 7.2 that comparison through process id to identify a FOSA process is not possible inside the create handler when overloading the *create_process()* function for container memory allocation. Therefore when initializing the user area variables inside the create handler the default boolean value on the *fosa_process* variable, indicating if the process is FOSA, is set to true. When the list iteration has been performed this value will be reset to false if the current process id was not found.

By an initial control inside the create handler, checking if the list is empty, we can eliminate some unnecessary list iterations. Since when a FOSA process is created a list node is added to the container we can avoid iteration for processes created before any FOSA process has been created. If the list is empty the process is obviously not of interest for execution time measurement since it can not be a FOSA process. Such processes are for instance system start up processes. Uninteresting non-FOSA processes created before any FOSA process will have their variable reset to false and will be returned from the create handler immediately.

If the list is not empty the process being created might be a FOSA process and the boolean value is kept true. In this case we need to save the process id and initial values in the user area. Finally we will also save the system clock tick length in user area (again to avoid global variables) to be used in later calculations.

If the list was not empty at the process creation the process could be a FOSA process, we need to compare the process id in the user area with those in the list nodes in order to determine if it is. Since this comparison will be made in the swap in handler, that might be called many times for one process, we wish to somehow check if the current swap in is the first one for that process. Obviously we do not want to iterate through the list every time the process is swapped in.

First as we enter the swap in handler check if the *fosa_process* variable has been set to false or if the list is empty. If not, continue. As the process is swapped in we need to fetch the system time and save in the user area as the slice execution start time. Now, check if the user area list node pointer contains a value other than NULL. If it does the list node has obviously been found before and a pointer to it has been saved in the user area. This meaning that it is not the first time in the swap in handler. In this case save the slice start time in the user area and exit the swap in handler.

If it is not the first time in the swap in handler, the list node pointer is null, we need to iterate through the list in order to find the list node for this process. It is not until now that we can identify a FOSA process. Simply, if the process id of the user area is not found in any list node reset the *fosa_process* variable to false and return. If a list node is found with the process id it is a FOSA process. Then keep the boolean value true and save a pointer to that node in the user area for immediate access to the process specific container position later on. Also save the start time of execution slice in the user area. Figure 7.4 shows the swap in handler implementation in a flow chart.

Finally let us have a look at the swap out handler where the time is actually measured. Now, we know whether the process is an interesting FOSA process or not simply by checking the *fosa_process* variable. The second step, if *fosa_process* is true, is to once again fetch the current system time. The slice execution time is calculated by subtracting the slice start time saved in the user area from this swap out system time. Since time is represented by a number of ticks and a number of microseconds since the last tick the latter subtraction might result in a negative value on the microseconds. In that case we need to decrease the number of ticks with one tick and recalculate the number of microseconds, see figure 7.5. By using the tick length, adding the negative number of microseconds to the tick length a positive representation is found.

When the slice execution time is calculated it will be added to the time of any previous slices stored in the list node. The total execution time for the process is then saved in the list node. Time will be represented in ticks and microseconds in the list node as well, we add the number of ticks to the list node tick variable and the number of microseconds to the list node microsecond variable.

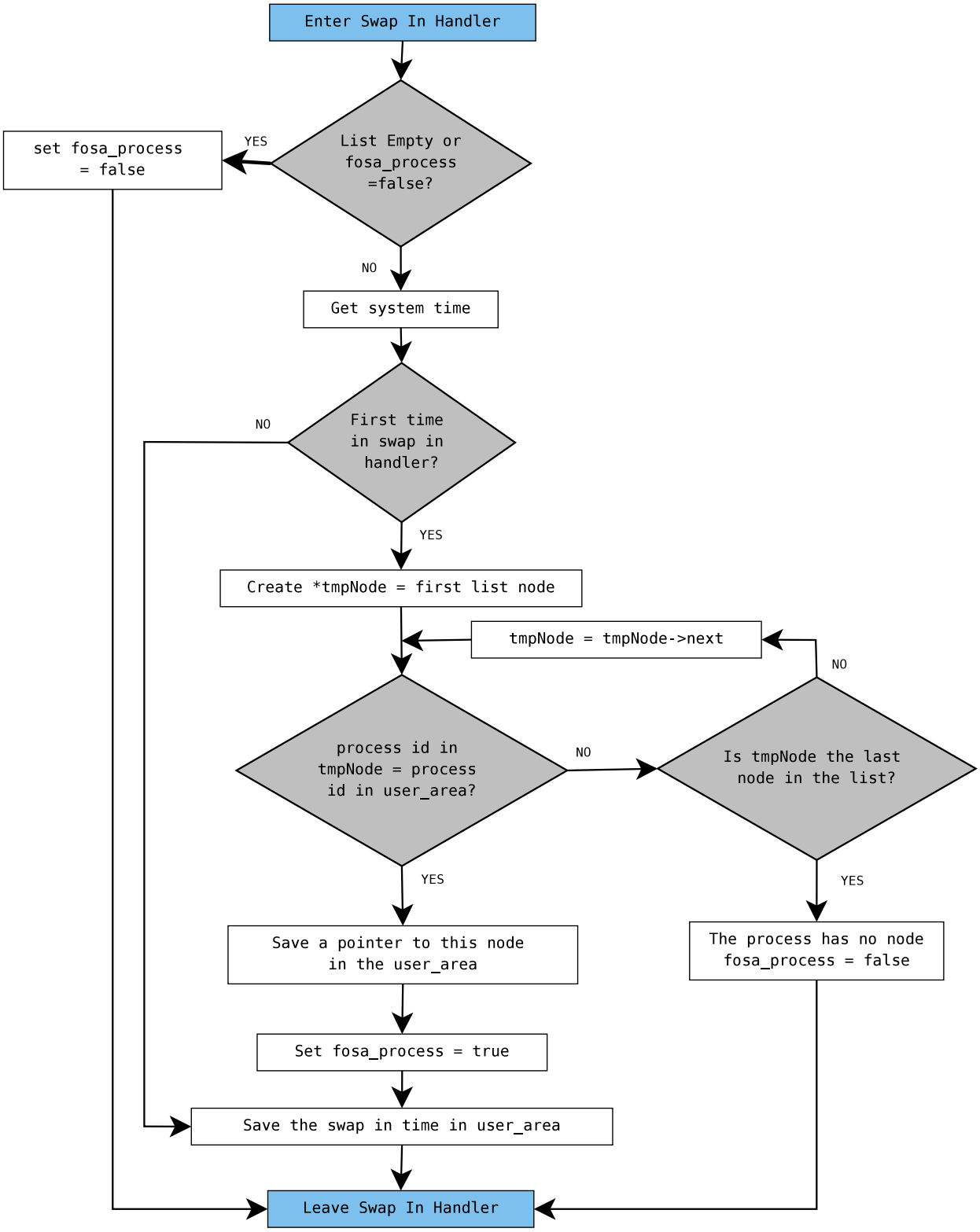
Enter Swap In Handler
List Empty or fosa_process =false?
YES
set fosa_process = false
NO
Get system time
First time in swap in handler?
NO
YES
Create *tmpNode = first list node
tmpNode = tmpNode->next
NO
process id in tmpNode = process id in user_area?
NO
Is tmpNode the last node in the list?
YES
YES
Save a pointer to this node in the user_area
The process has no node fosa_process = false
Set fosa_process = true
Save the swap in time in user_area
Leave Swap In Handler

Figure 7.4: Swap in handler flow chart.

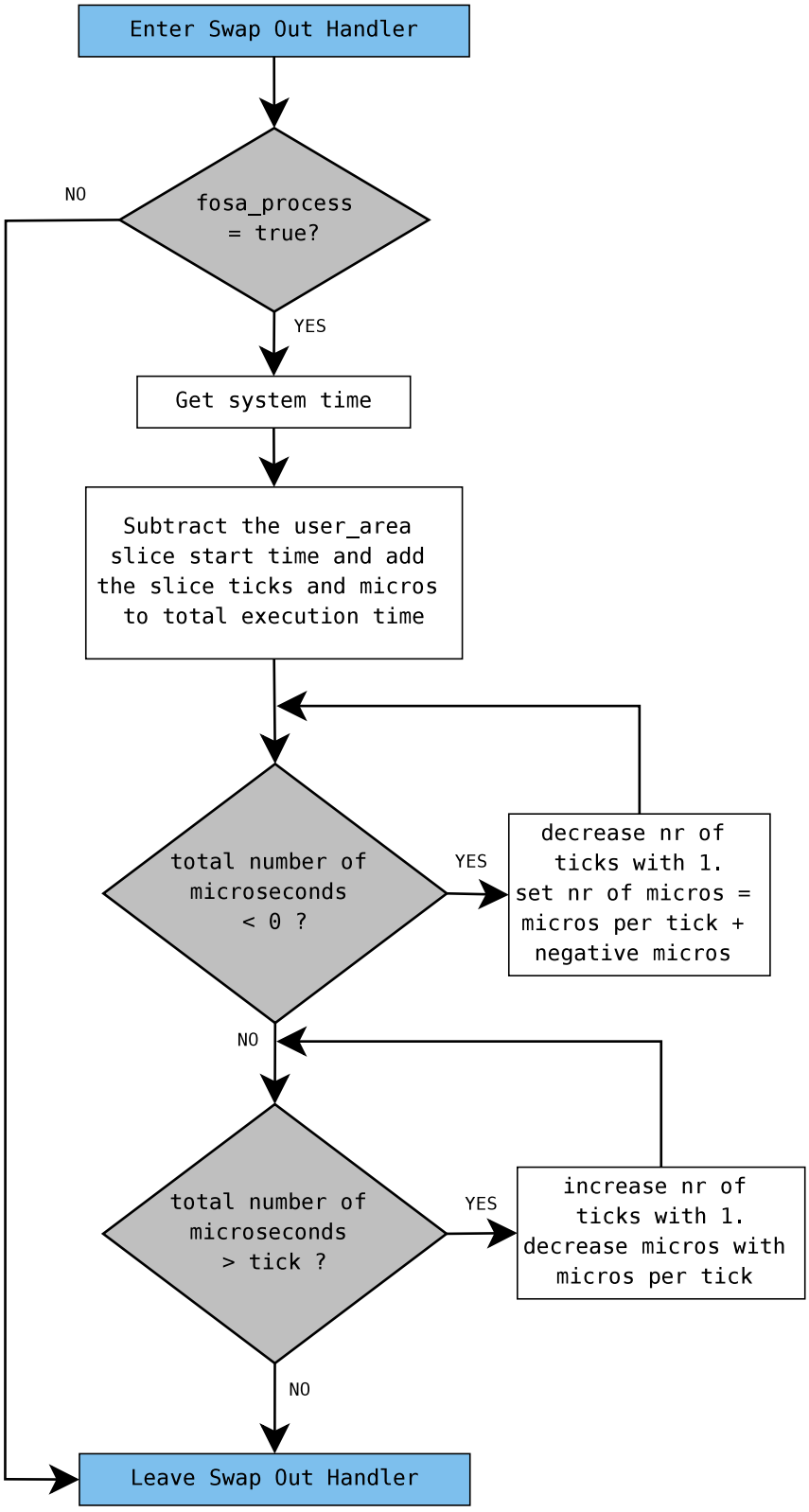

Figure 7.5: Swap out handler flow chart.

7.6 Test Application

To verify the correctness of the execution time calculation algorithm a software test designed. The purpose of this test is to recognize context switching of FOSA processes and to ensure correct execution slice time calculations. The test should also ensure that the correct result is stored in the list node and that processes other than FOSA processes are ignored.

At least two FOSA processes should be created with different priorities and some delay to cause context switching. The main test process will be a static prioritized process, PRI_PROC, automatically created when the application is started. Such processes should be defined in a main application configuration file, "osemain.con" in the project source code folder, see appendix E. The file is essential in order to inform OSE about which processes that form the main application of a program [8]. This main test process will create and start the other two processes. If they are given a higher priority than the main function they will be preempted as soon as they are stated.

Since there is no available create FOSA process function the create process function must be overloaded to simulate one. A list node should be created at a FOSA process creation to store execution times for the respective FOSA process. When the results are no longer needed the list node memory should be freed. It is reasonable to believe that the measurement results will no longer be needed after a process has died, however the result might be needed just as the process has finished execution and therefore to be safe node deletion will not be performed in a kill handler. Instead a function to delete nodes will be designed for the test application to call. See appendix B for the create and delete functions.

The test will be simple, three processes will be created. First one FOSA process, then another FOSA process of higher priority and finally a non-FOSA process of even higher priority than the last one. The two FOSA processes will contain a loop performing some operations, simply to consume time, and after a certain amount of operation there will be a delay. Process one will be invoked first but dispatched when process two is ready, process two will then be dispatched as the non-FOSA process is ready. The non-FOSA process contains nothing and finishes immediately. Process two will then perform a preemption to reqire the CPU and run until it reaches a delay, process one will preempts and runs until delay and so on until process death.

To display the results the ramlog is utilized. The ramlog is a memory area in the RAM that platform and applications can write trace and debug messages to. It is designed to survive system crashes so that the crash reason can be analyzed [10]. By the call *ramlog_printf()* writes are performed. Running the reference system RTOSE the *rld* call displays the ramlog. By printing slice start times, swap out times and the calculated slice execution time the functionality of the slice execution time calculation can be verified. By printing the previous result stored in the node and the new value after addition of the last slice execution time the final result will also be verified.

By printing the process id of a process being swapped in when a list has already been created, "possible FOSA process" swap ins will be detected. If a node is not found and the FOSA process variable is set to false print that a non-FOSA process was detected and ignored. Adding a print command last in the swap out handler will show when a FOSA process only has been swapped out.

7.7 Software Test Result

An extract from one of the ramlog prints is seen in figure 7.6. Obviously the code has been refined until correct measurement functionality is achieved. Seen in figure 7.6 the slice start time stamp of process one is 276 ticks and one microsecond, the end slice timestamp is 276 ticks and three microseconds. The difference between them is definitely two microseconds and the calculation has obviously been performed correctly. The same comparison for other execution slices support this conclusion as well.

As predicted process one starts executing and when process two is ready it will preempt. However, comparing the process one dispatch time with the process two preemption time it is revealed that processes two being ready is not the reason for the dispatch of process one. This is not a problem, actually it is reasonable that process one having such low priority is preempted by some other process in the system. That preempting process is not a FOSA process since it is being ignored, no execution time calculation is performed and hence no ramlog prints.

From this ramlog it is also concluded that a non-FOSA process created when the list is not empty will still be detected as a non-FOSA process. Any such process will be ignored from execution time measurements.

```
Process1 created with id 10041 and priority 24

First SWAP IN of 10041 (possible FOSA) at t=276 us=1
        10041 identified as FOSA process
DISPATCH FOSA process: 10041 at t=276 us=3
        Added slice time to NODE, current value for 10041 t=0 us=2

Process2 created with id 10042 and priority 16

First SWAP IN of 10042 (possible FOSA) at t=277 us=0
        10042 identified as FOSA process
DISPATCH FOSA process: 10042 at t=277 us=2
        Added slice time to NODE, current value for 10042 t=0 us=2

Non-FOSA process created with id 10043 and priority 3

First SWAP IN of 10043 (possible FOSA) at t=277 us=3
        NO node found, 10043 is not a FOSA process => IGNORE process

PREEMPT - FOSA process 10041 at t=277 us=4
DISPATCH FOSA process: 10041 at t=278 us=5
Added slice time to NODE, current value for 10041 t=1 us=3
```

Figure 7.6: A part of the ramlog print from a test performed in the soft core environment.

As desired the slice time calculations and summary in the list nodes are correct. Both the case of recalculation for a negative number of microseconds and a number of microseconds that is larger than one tick has been verified [1], however those cases are not displayed in figure 7.6. Context switches including FOSA processes are detected and thereby execution time for the correct processes are measured.

[1] Even though the calculations are performed correctly in this case the following time stamp reported by get_systime() turns out wrong. This is, according to Mathias Bergvall [43] most likely due to a bug in the softcore BSP.

Chapter 8: Verification

According to the software test results the execution time measurement has succeeded in the soft kernel environment on the Intel x86 processor. Still, verification is necessary in order to determine the time overhead and memory utilization on a real embedded processor. For this purpose an application development system from Freescale Semiconductor, M9328MX21ADSE, will be used. The i.MX21 board contains an M9328MX21 application processor with an ARM926EJ-S core. There is no particular reason for choosing this hardware configuration more than that it is possible to use it for verification of the execution time measurement.

Running RTOSE with the execution time measurement core module on the ARM9 processor will show the performance of the measurement function on a real embedded processor. Beside verifying the results from the softcore tests it will also be possible to determine the time and memory overhead caused by this measurement functionality. This will decide whether the implemented solution is realistic for future use with the dynamic scheduler.

Initially in this chapter there is a detailed description of the hardware development environment and instructions on how to load the RTOSE image to the ARM processor. Conclusions will be drawn regarding memory utilization and two tests will be designed to measure the extra context switching latency. Finally the test results will be presented and in chapter 9 these results will be used to draw final performance conclusions.

8.1 Hardware Tools and Environment Configuration

As in the softcore environment configurations are needed for the hardware environment to implement the tests. Both installing new programs and setting up licenses is required and likely new hardware will be needed on the host PC. There are alternative ways to communicate with the i.mx21 target board and processor, over Ethernet or through a universal asynchronous receiver/transmitter (UART). Settings for the chosen communication method should be made in the target configuration file.

The latter alternative using a UART serial port was chosen in this case. First of all a serial port must obviously be available on the host PC. Then, when the target board is connected to the computer you will need a terminal program to display the information arriving on the serial port. One such program is UTF-8 Tera Term Pro free to download at the Tera Term homepage [22]. Start Tera Term Pro and choose serial communication on the connected port. Then set the baud rate, the speed of transmission through the cable, to the rate at which the board is sending information. Go to setup and serial port then choose 115200 Bd. This baud rate for the i.mx21 board is specified in the board configuration file, rtose.conf, found in the refsys/rtose/mx21 folder.

Now, using the terminal window to watch serially received data, power on the processor. Assuming that there is already a bootloader available in the flash memory loaded at start familiarize with available commands. The bootloader commands can for instance be used to load the desired rtose image to the processor.

One suitable bootloader to use would be POLO, the second OSE reference system mentioned in section 7.1. In this case however the bootloader available in flash is BLOB [24]. BLOB is bootloader made particularly for booting the Linux operating system. Changes in the configuration and make files for the mx21 board are necessary in order to use the blob bootloader for booting OSE. In the makefile rtose.mk change the image start so that OSE will be placed at the same address as where Linux is expected. That address would be 0xc0008000 [42]. Be careful not to make changes causing memory areas to overlap. As the image file to be loaded has been created consider the size of that image when configuring the memory sizes. Overlapping memory areas will cause a fatal early system start up error.

The image file for the mx21 board is created precisely as for the softcore. Start cygwin, go to refsys/rtose/mx21 instaed of refsys/rtose/sfk-win32. If configuration settings have been made and makefiles are modified to include the desired modules run the make command. Either the terminal program or a debugger can be used to load the image file to the target processor. Using BLOB the file can be loaded through TFTP, the Trivial File Transfer Protocol. In order to do that a TFTP server needs to be installed on the host PC, free servers are available on software download sites. Place the image file in a tftpboot directory in the TFTP server folder. Set the ip-address for your server and the target board, point out the file to be transferred and TFTP it. See the example below.

```
server 10.0.0.1
ip 10.0.0.2
Tftpfile tftpboot/image.bin
tftp
```

If the transfer worked properly type *call 0xc0008000* to start the OSE image from the image start location. With BLOB do not try the boot call unless your image is Linux. As rtose starts all the same commands used at the softcore simulation is available, for example the ramlog diplay command *rld*. That way, if you have one working image without the extra module, previous ramlogs can be viewed to find information about a crash. However, a better alternative would be to use a debugger for error detection.

In this project the LA-7702 debugger from LAUTERBACH Datentechnik[23] is used with a LA-7742 JTAG cable for ARM9 processors. The accompanied software is TRACE32 to be installed to the installation guide accordingly. There are licenses required both for the ARM9 debugger and for the TRACE32 software. How to set up these licenses is described in [17]. The lauterbach debugger can either be connected through a parallel port as in this case or through ethernet. Using TRACE32 and the debugger it is possible to view register contents, set breakpoints and detect errors.

To load the image file through the debugger TRACE32 commands are used. Typically a scrpit file is compiled to include for instance processor specification and the load commands. The script file could include initialization of the target board but in this case the bootloader BLOB performs this initialization. That means that BLOB has to be started before the image file is loaded. Since when using BLOB you have to interrupt the automatic boot of Linux to start the bootloader Tera Term will be used to do so.

1. Start TRACE32.
2. Power on the debugger and target board.
3. In TRACE32 go to setting CPU, chose the M9328MX21 processor and connect.
4. START Tera Term.
5. Press GO in the TRACE32 List Source window.
6. Fast, stop the automatic Linux boot from the Tera Term window.
7. Open and run your script file in TRACE32
8. Run your rtose image file by pressing GO once again in the List Source window.

The list source window of TRACE32 displays the current address position. At the first GO this address should be all zeroes and at the second GO the address should be that of the rtose image start. Another useful window in TRACE32 is the "Stackframe with locals" window, if an error occurs this window displays the error code. Error codes in OSE is constituted of two parts masked together. The first part tells what kind of an error that has occurred and the second part specifies the cause. For example are errors is 0x8000000 the fatal error mask. Some error sub codes found during the project hardware adaption were:

```
0x114     OSE_MM_ECONFIGURATION_ERROR
0x115     OSE_ESYSCALL_TOO_EARLY
0x113     OSE_EUNEXPECTED_EXCEPTION_REGDUMP
```

The first error occurred as mentioned previously when there were overlapping memory areas in the configuration. Which areas that overlap can be found in the ramlog. The second error occurs if a system call is made before the system start up is finished. The last error was due a faulty pointer usage in the execution time measurement module.

8.2 Time Overhead Test Specifications

To determine whether the quality of this solution for execution time measurement is adequate the caused time overhead or context switching latency must be tested. In this purpose two different tests has been designed. Both tests will investigate the difference in time when executing with the measurement functionality and without it. That way the impact of the measurements on the system execution time will be recognized. Meaning these tests should determine:

- The context switching latency.
- The impact of this latency on the system performance.

One of the two tests is a modification of a test module included in the OSE installation, called pbench. The pBench module is designed to benchmark OSE system calls [10]. Meaning that it is a test of which the results will serve as a reference. The pBench module measures the time it takes to perform different system calls on the target hardware. For example calls to allocate memory or to create a process.

The one pBench test of interest is the *send+swap+receive* test. It measures the time to perform one swap as a signal is sent from one process to another. This test will be modified to include a FOSA process and the designed swap handlers. The pBench module will perform the measurement for this call a hundred times. In that way the min, max and a mean value can be presented.

The second test is designed to determine how frequent context switching of FOSA processes in a system will affect the average performance. In similarity with the pBench test swaps will be achieved through send and receive of signals. A desired number of swaps for the test will be specified and two processes will be designed decreasing this value each time they are swapped in. Further on using any reference to one of the tests, this test called "test1" is the one meant unless pBench is specifically mentioned.

The pbench module will perform tests swapping between one FOSA process and one regular processes while the second test will be swapping between two FOSA processes. That way in the second test both the customized swap in and a swap out handler will be invoked at each swap. In the pBench test as only one process is a FOSA process one process will be ignored from measurements.

8.3 Test Implementation of Test1

The key idea of this test is to run an application with customized kernel handlers and then run the same application without them to compare the execution times. The difference in time is expected to be relative to the number of context switches performed since the only functional difference is the context switches. The more swaps the larger difference in execution time is expected.

When designing an application for this purpose one needs to be careful. Remember that the execution time of a system call may vary and therefore an application implementing such a call would not perform identically in a second run. The application execution time must be constant in order for comparison to be reliable.

There is another important challenge designing the test application. In order to discover the context switching latency the processor must be 100% utilized when running the application. If there are periods where nothing needs to be done longer than the total context switching latency this latency will not affect the system performance at all. It will only result in a higher utilization of the processor. When the processor is 100% utilized running the test application the added context switching latency using the kernel handlers will cause a system slow down. It is that worst case slow down that is of interest here.

Running the application with the customized kernel handlers might not result in the same number of swaps as when running without it. In order for the two tests to be comparable the number of swaps must be known. These complications to the simple initial idea motivates a modification of the design requirements with the mentioned parameters.

- The test processes should include only operations of constant time.
- The application should utilize 100% CPU.
- The tested number of swaps should be controllable.

```
OS_PROCESS(process1){

   union SIGNAL *sig;

   while(1){
      sig = receive(sel_swap);
      sig->swpsig.count--;

      if(sig->swpsig.count == 0){
         send(&sig, test_application_main_);
         kill_proc(current_process());
      }
      else
         send(&sig, sig->swpsig.p0);
   }
}
```

(a) Test process.

```
#define SWAP_SIGNAL 1000

struct Swap_signal{
    SIGSELECT sig_no;
    PROCESS p0;
    PROCESS p1;
    unsigned long count;
};

union SIGNAL{
    SIGSELECT sig_no;
    struct Swap_signal swpsig;
};

static const SIGSELECT sel_swap[]
   = { 1, SWAP_SIGNAL};
```

(b) Signal definition.

Figure 8.1: Test process structure and signal definition.

The test processes will be given highest priority in order avoid unnecessary preemption by other processes. For full processor utilization delays should be avoided, to ensure preemption without delay signals will be used. Unfortunately the send and receive calls are not constant in time but this is still likely to be the better alternative. Sending signals back and forth enables swaps counting and control. The application will basically not include anything but swapping the two processes.

As the signal is sent from one process to the other a predefined swap count integer is decreased. A test process and the signal definition code is shown in figure 8.1.

The test process with entrypoint process1 is displayed in figure 8.1a. It waits to receive a signal of the *Swap_signal* type defined in figure 8.1b. The signal contains a signal number, a count variable and two process ids. As process1 with id p1 receives the signal it is swapped in, it will then decrease the swap count variable. Unless the count variable is zero process1 will send the signal further to process0, with id p0. The structure of process0 is identical to that of process1 sending the signal back to process1 after a swap count decrease.

The test is initiated by a function *run_test()* called by the main process. It sends a *Swap_signal* signal to one of the test processes and as the swap count variable reaches zero the current test process will send the signal back. This part of the run test process is shown in figure 8.2.

```
static const SIGSELECT sel_swap[] = { 1, SWAP_SIGNAL};
union SIGNAL *sig1;

sig1 = alloc(sizeof(struct Swap_signal), SWAP_SIGNAL);
sig1->swpsig.p0 = p0;
sig1->swpsig.p1 = p1;
sig1->swpsig.count = points*NR_OF_COUNTS;
send(&sig1, p0);

sig1 = receive(sel_swap);
if(sig1->swpsig.count != 0)
    ramlog_printf("Error count != 0\n");
free_buf(&sig1);
```

Figure 8.2: Part of the *run_test()* function.

The signal type is selected and memory is allocated for it. The desired number of swaps to be performed is specified in the count variable and the ids of the two test processes are saved in the signal. The signal is sent to process0 starting the test. When the signal returns through the receive call the test is finished. Figure 8.3 illustrates how this test is performed.

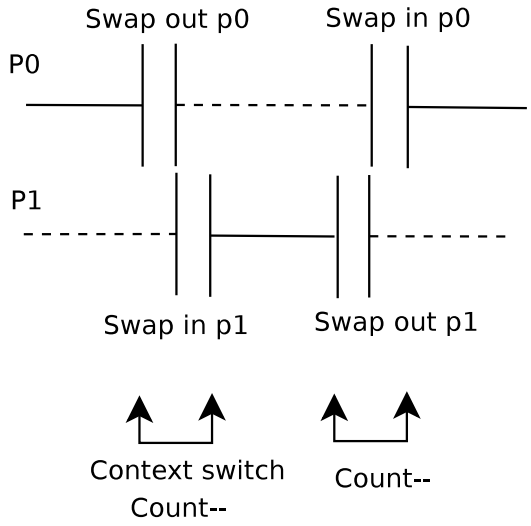

Figure 8.3: Illustration of test1 measurements.

Initially inside a non FOSA process, referred to as the main process, time is measured and the *run_ test()* function is called. The signal is sent to process0, p0 in the figure, which sends a signal to process one and a swap occurs. At each swap between the two FOSA processes p0 and p1 the count variable is decreased. When the variable equals zero the test is finished and the current process sends a signal back to the main process. Time is then measured once again and the difference in time is calculated. This calculation of application execution time will be preformed many times with and without kernel handlers for different number of swaps (different values on the count variable). The result is a number of points (time, swaps) to be plotted and compared in a graph. One curve will represent the kernel handler implementation and the other one will show the resulting time without it.

8.4 Test Implementation of pBench

The pbench module includes a file *sendswap.c* responsible for the previously mentioned *send+swap+receive* test. The function initiating the test is called *measure_ send_ swap_ receive*. It is called by the main test process located in the *pBench.c* file. This file also includes functions for handling the results stating the min, max and mean values as well as the standard deviation.

Inside the test function a process will be created and a signal will be allocated. This test will be modified by making the receiver process with id pp1_ a FOSA process. Instead of calling the *create_ process()* function in *measure_ send_ swap_ receive* the overloaded version creating a list node for execution time measurements will be called. This means that at each context switch one regular process and one FOSA processes is swapped. Figure 8.4 and 8.5 illustrates this modification.

```
pp1_ = create_process(OS_PRI_PROC, "receiver2", receiver,
                      512, get_pri(current_process())-1, 0, 0, 0, 0, 0);
```

Figure 8.4: Before modification of the *measure_ send_ swap_ receive* function.

```
pp1_ = create_test_process("receiver2", receiver,
                get_pri(current_process())-1);
```

Figure 8.5: After modification of the *measure_ send_ swap_ receive* fucntion.

The created pp1 process has higher priority than that including the test function. This means that as a signal is sent to the pp1 process a context switch will occur. The allocated signal will be sent from the function to the pp1 process one hundred times. The time it takes to perform this context switch will be measured at each iteration by an implemented a hardware timer. Results will be stored in an array to be processed later, figure 8.6a and 8.6b shows this design.

```
LOCK_SAVE(msr);
   for (i=0; i<*num_iter; i++){

      SIGSELECT ss[] =
      {1, MEASURE_RESULT_SIGNO};

      CLEAR_TICKS();
      send(&sig, pp1_);
      sig = receive(ss);
      results[i] = sig->res.ticks;
   }
LOCK_RESTORE(msr);
```

(a) Part of the test function.

```
while (1){

   sig = receive(any);
   ticks = READ_TICKS();

   if (sig->sig_no == MEASURE_RESULT_SIGNO){
      sig->res.ticks = ticks;
      send(&sig, sender(&sig));
   }
   else
      free_buf(&sig);
}
```

(b) Part of the pp1 process.

Figure 8.6: The pBench *measure_ send_ swap_ receive* test.

The modification of the test making the pp1 process to be FOSA will consequently only be measuring the context switches where a regular process is swapped out and a FOSA process is swapped in. In that way the additional time caused by the swap in of a FOSA process can be held by subtracting benchmark values from the new results. The max time value, corresponding to the worst case swap in time duration will indicate the time spent due to list iteration. However, in this test only one FOSA process is created and therefore there is only one possible node to search. Consequently this list "iteration" is the fastest possible.

Figure 8.7 illustrates a context switch between two processes, p0 and p1. Assume that p0 is not a FOSA-process while p1 is a FOSA-process. The figure shows how the non FOSA process is being swapped out and the FOSA process being swapped in. In the pBench test a time stamp is requested by p0 before the sending the signal to p1. P1 then requests a time stamp as it is swapped in. The time difference representing a send+swap+receive is calculated and the result is saved. This is repeated 100 times, no measurements are made in the opposite direction as the FOSA process is swapped out and the non FOSA process is swapped in.

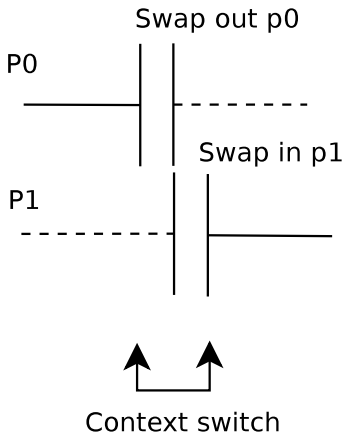

Figure 8.7: Illustration of send+swap+receive measurement.

8.5 Hardware Test Results

First consider the results from the pbench test without modification. Not using any FOSA process or kernel handlers will give the default time of a context switch. These values will as suggested be used as a benchmark for comparison with test results from measurements when execution time measurement is invoked. The pBench measurements are reported to be of a timing precision at 0.092 microseconds. This resolution is a result from the i.mx21 hardware timer utilization. It obviously presents a higher resolution than when using the *get_ systime()* command. Consequently the pBench test is more suited for measuring the exact time of one swap than test1 is. Test1 will be used to review results at a large amount of frequent context switches.

The pBench test will provide a median value for the extra time caused when using the swap-in handler. In test1 the results will be used to calculate the mean value of this time over a large amount of swaps. This test will also calculate the mean time overhead when FOSA processes are being swapped both in and out at the same context switch.

By using the shell command defined in the main configuration file, oesmain.con, of the pBench module the test will be run. Choosing verbose mode with the *-v* argument extra information compared to default will be displayed. Simply type the *pBench -v* command in the cygwin window as RTOSE is running on the processor. Table 8.1 presents a table displaying the results from three repeated *send+swap+receive* measurements when no execution time measurement is performed. These benchmark values will be used for comparison with later results.

Measurement	Median [us]	Std.Dev.	Min [us]	Max[us]
1	6.37	0.12	6.28	12.20
2	6.28	0.15	6.19	10.26
3	6.37	0.10	6.28	12.29

Table 8.1: Benchmark pBench measurement results.

As seen from the table a default context switch on the arm9 processor usually takes somewhat more than six microseconds. In the worst case the context switching takes up to twelve microseconds. The median time value of a context switch is nearly the same as the minimum value. These results indicate that this time value is most likely the context switching time in most cases. Rare and rather large deviations from this median value causes the maxi-

mum value. Several pbench measurements have been made using the modified test include a FOSA processes and kernel handlers. Table 8.2 show the resulting values from ten such measurements.

Measurement	Median [us]	Std.Dev.	Min [us]	Max[us]
1	14.60	0.53	14.23	20.05
2	14.60	0.39	13.86	20.14
3	14.51	0.47	14.04	20.05
4	15.06	0.50	14.32	20.70
5	14.69	0.39	13.95	20.42
6	14.60	0.40	14.14	19.86
7	14.69	0.69	14.14	110.31
8	14.69	0.57	13.86	57.47
9	14.60	0.55	14.14	20.51
10	14.69	0.54	13.95	54.51

Table 8.2: Measurement results from pBench using a FOSA process and kernel handlers.

In similarity with the benchmark table values the min and median values are real close to each other. Comparing to the benchmark results the median time seems to have increased about 8 microseconds. The increase in this case corresponds to the extra time when a non-FOSA process is swapped out and a FOSA process is swapped in. That would indicate 8us extra only due to the swap in handler. Evidently the resulting value is more than twice as large as the original one. The max value sometimes become relatively large and this can not be due to a long list iteration since there are only one list node created. Instead this is probably because of the priority level on the processes in this test. It is likely interrupted by a higher priority process between the time measurements.

In test1 a large amount of swaps are performed, the maximum possible swapping frequency is used in order to fully utilize the CPU [1]. Producing the results shown in figure 8.8 application time measurements have been made nine times for a different number of swaps. The same measurements are made for the case when using the kernel handlers, when not using any kernel handler as well as when only the swap-in handler is used. The number of swaps reaches from one hundred thousand to nine hundred thousand. These results will show the caused time overhead as swaps are many and extremely frequent.

As expected the time overhead increases with the number of swaps. The actual values producing the plot in figure 8.8 are displayed in table 8.3. These values can be used to calculate a mean value of the extra context switching latency. The calculated mean value resulting from the measurement when only using the swap-in handler will be compared to the median value from the pBench test.

If an application will swap in and out FOSA processes one hundred thousand times the extra context switching latency will be 331ticks and 3836 us. Swapping FOSA processes nine hundred thousand times would cause a delay of 2938 ticks and 2694 us.

[1] If the CPU is not fully utilized the time difference to perform a certain amount of swaps in the two cases will be misleading.

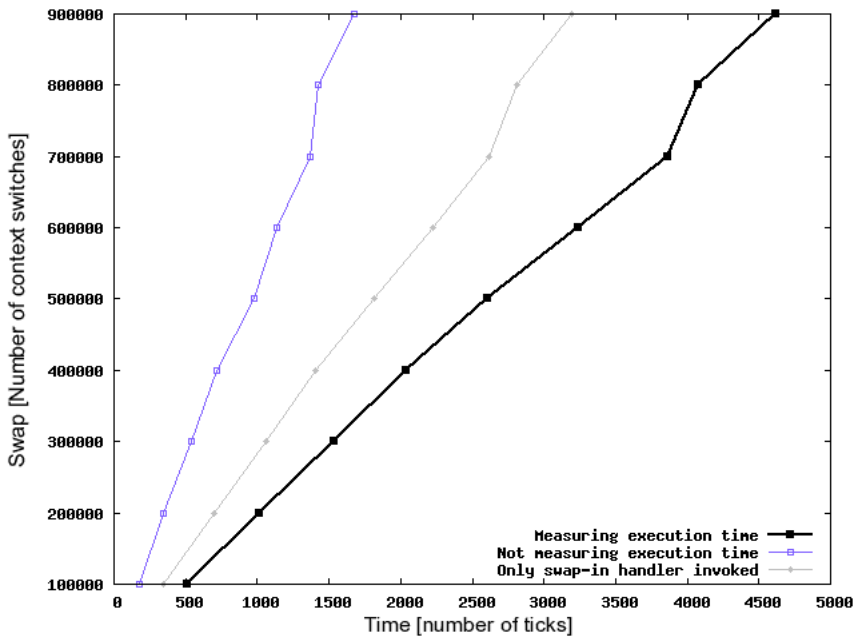

Figure 8.8: Time relative to number of swaps with and without execution time measurement.

Nr.of Swaps	Not using any KH	Swap-in KH only	Using swap KH
100000	178+0063	340+3664	509+3899
200000	345+0250	702+1073	1018+3464
300000	542+3059	1063+2336	1536+2788
400000	714+3646	1404+0283	2037+3527
500000	977+2302	1816+2685	2610+2035
600000	1133+0415	2229+1283	3241+3267
700000	1373+1153	2615+3995	3860+1489
800000	1428+2032	2808+0653	4080+3483
900000	1675.2182	3191+3981	4614+0876

Table 8.3: Measurement results [ticks+us] making the figure 8.8 plot.

The mean value of the context switching time when swapping two FOSA processes can be calculated from the table values. At 100000 swaps the mean swap time is 20.4us/swap [2] and at 900000 swaps the mean time per swap is 20.5us/swap [3].

The mean swap time when not using execution time measurement at 100000 swaps is calculated in the same way to 7.1us/swap and at 900000 it is 7.4us/swap. The difference in between these values gives the extra context switching latency due to the execution time measurements when FOSA processes are both swapped in and out. That would be 14.3us/swap and 14.1us/swap respectively.

[2] Total time: 509*4000 + 3899 = 2039899 us, 4000 being the tick length on i.mx21
Mean context switching time: 2039899/100000 = 20.4us/swap
[3] 4614*4000 + 879 = 18456879us, 18456879/900000 = 20.5us/swap

The mean value is obviously about the same, about 14us, independent of the number of swaps. Only using the swap-in handler and not the swap-out handler at 100000 swaps gives 13.6us/swap in total swap time and at 900000 14.0us/swap. A subtraction of the values from not using execution time measurement gives a swap-in time of 6.5us/swap and 6.6us/swap respectively. The influence of the swap-out handler is therefore about the same or slightly more, about 14us - 6.5us, than that of the swap in handler. Using both kernel handlers give an extra swap time of about 14us of which the swap in handler is responsible for about 6.5us and the swap out handler for about 7.5us.

Comparing with the pBench test results the mean values from test1 are slightly lower than the median values found from the pBench test. The median time of a swap in the pBench results showed a value of about 8us extra latency, here using mean values it is found to be about 6.5 us.

Chapter 9: Performance Evaluation

The test results presented in chapter 8 is now to be evaluated. To conclude whether the extra context switching latency and memory consumption can be acceptable application assumptions must be made. In this chapter such assumptions are made and the performance is evaluated with that ground. There will be a discussion on how to draw reasonable conclusions from the test results.

Section 9.1 discusses the evaluation ground and assumptions. Section 9.2 states the effects expected from memory utilization and section 9.3 discusses the influence of the extra context switching latency on a real system.

9.1 Evaluation Grounds

In order to draw conclusions on how the execution time measurement functionality affects the average system performance, the amount of extra latency acceptable must be known and compared to the increased context switching latency. It is not reasonable to only compare the context switching time without considering any application. The total latency in a system of course depend on the switching frequency. The resulting extra latency due to measurements in the kernel handler must be calculated multiplying the extra latency per context switch with the number of switches performed.

Consequently the switching frequency must be known in order to find the latency. Since this design is not intended for any particular application or switching frequency the quality in performance can only be estimated from assumed conditions. There is no standard swapping frequency but according to Magnus Karlsson [42] a probable interval would be between zero and 1000 swaps per second. He also mentions that in a unix system for example there are about 50 swaps per second. Consider 1000 swaps per second a maximum swapping frequency that will result in a maximum measurement latency.

Concerning the memory utilization, in order to consider the extra allocated memory space due to the measurements the number of processes concurrently active in the system must be know. The user area memory is allocated for every process in the system and deleted at process termination. The container memory for storage of the measurement results is an additional memory space for every FOSA-process in the system. In order to find the exact amount of memory consumed the exact amount of both processes and FOSA-processes in the system must be known.

There is no standard for the number of processes in a system either, it is different for each application. Since the FRESCOR scheduling framework is mainly designed for improving resource utilization in soft real time systems assume that the application is a telecommunication system. Mathias Bergvall [43] states that in such a system the number of processes probably reaches between 100 and 1000. He also mentions that in a safety critical system it will be less than 100 processes. Magnus Karlsson declares that the number of processes will hardly ever be less than 30. Assume that 1000 processes in a system is the maximum possible number of concurrent processes resulting in the maximum amount of memory consumption.

Also assume that a majority of these processes are FOSA-processes whose execution time should be measured. Any application process would be a FOSA-process negotiating contracts with the scheduler. Only OSE system management processes are likely not to be FOSA processes.

9.2 Memory Utilization

The user area is designed to give each process 21 bytes extra process specific memory. The allocated memory size of a list node is easily found from the *sizeof(fosa_ose_list_node)* call, or by summarizing the node variable sizes. In this case it is 20 bytes. How much of the available memory that is consumed as there is a certain number of concurrent processes in the system can be found from the following formula.

$M[Bytes] = N * 21[Bytes] + NFOSA * 20[Bytes]$

N is the total number of concurrent processes and NFOSA is the number of FOSA processes ever created in the system of which the node has not been deleted. In the worst case of 1000 concurrent processes in the system, assuming that all processes are FOSA processes and no node has yet been deleted, the consumed memory space would be 41kB [1]. Whether 41kB extra memory allocation due to the measurements can be approved depends of course on the memory space required by the application, and the total amount of memory available in the system.

9.3 Time Overhead

From the results in section 8.5 the extra context switching latency due to the kernel handler implementation of execution time measurement is determined to 14us per context switch, if both the swap in and out processes are FOSA-processes. This result can be used to determine the influence of this latency at different swapping frequencies. As mentioned in section 9.1 the swapping frequency can be expected to be between zero and 1000 swaps per second [42]. A calculated extra latency per second can therefore be found for each frequency by multiplying the latency with the number of swaps per second at that frequency. Doing so for frequencies ranging from zero to 1000 results in the graph in figure 9.1.

At a frequency of 1000 swaps per second the extra latency will be 14 milliseconds. This means that if the processor is fully utilized (100% CPU) then the application will be delayed 14 milliseconds per second of execution, assuming that the same number of swaps will be performed under the new conditions. For example a service that would normally take one second would now take one second and 14 milliseconds. Similarly a service that would normally take one minute would now take one minute plus 0.84 seconds.

However, if the processor was 100% utilized a scheduling framework for more efficient resource utilization would not be needed. In fact since the purpose of the FRESCOR project is to distribute spare capacity the processor of a system to which this scehduling framework should be applied is definitely not already fully utilized. As the latency increases with 14ms in the worst case assuming full processor utilization, it increases from about 6ms to 20ms at

[1] 1000*21 + 1000*20 = 41000

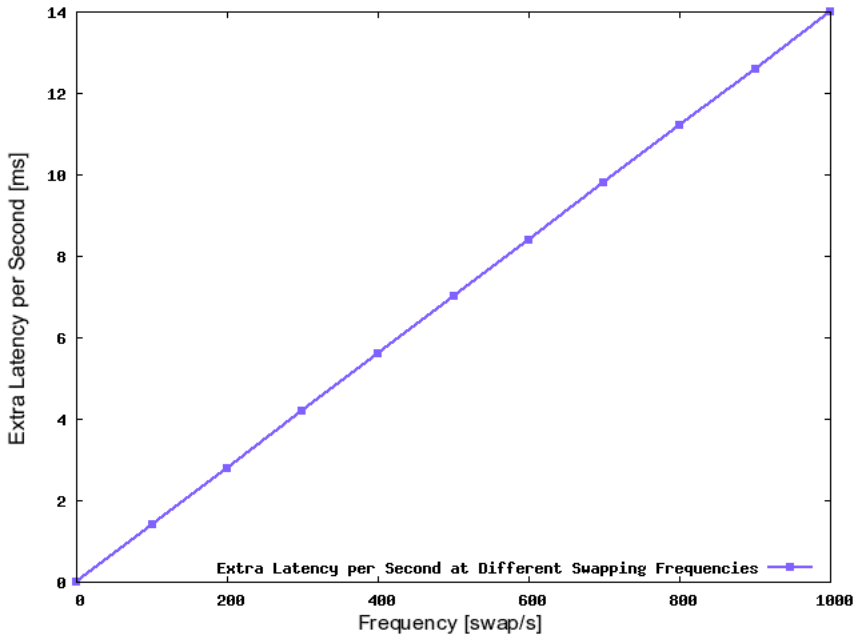

Figure 9.1: Extra context switching latency due to measurements per second at different swapping frequencies.

a swapping frequency of 1000 swaps per second in one second execution. Consequently the context switching time goes from 0.6% [2] of the service execution time to 2% [3] of the service execution time.

Obviously the context switching latency increase is about 200%. Still the total service execution time increase is only 1.4% in the worst case when the context switching time is 2% of the service time. In this case if the service time is originally one second the total execution time will be increased with 14ms which corresponds to a 1.4% increase. This increase is a small part of the service execution time that is likely to be covered by a small part of the spare capacity.

Note that these numbers are found from tests on the ARM926EJ-S processor core operating at a maximum frequency of 266MHz [20]. When using a slower processor the latency would of course be longer then 14us/swap and with a faster processor it would be less. The i.MX21 development kit was not chosen with any particular application in mind that would be suitable to run on the ARM926EJ-S processor. However the i.MX21 development board is optimized for portable multimedia applications [20], being typical soft real-time systems which is also the target of the FRESCOR scheduling framework.

Whether the extra context switch latency can be allowed without having a negative effect on the system performance will depend on the application, especially concerning requirements on response time. It is however unlikely that the extra latency will cancel out the benefit of using spare capacity assuming that the spare capacity capabilities will free more resources than what is worth 1.4% in execution time. It is therefore reasonable believe that the kernel handler implementation method could be used for some application purposes to assist the FRESCOR scheduler with process execution time measurements.

[2] 6ms/1s = 0.006, 0.6%
[3] 20ms/(1s+14ms) = 0.0197, about 2%

Chapter 10: Conclusions and Future Work

To declare whether the implemented method for execution time measurement is sufficient or not for usage with the FRESCOR scheduler for a general application is not possible. However with a specific application in mind, memory and time overhead characteristics can be evaluated and the suitability of this measurement solution for the particular application can be determined. None of the other two evaluated methods, not the RMM or kernel modification method, will present measurements more accurate or reliable than the kernel handler method.

If the response time and memory consumption requirements from the application can not be fulfilled when using this kernel handler measurement implementation, only improvements of this very same method can serve as a realistic alternative. Such improvements for speed, accuracy and memory consumption could possibly be:

- Let a more experienced programmer simplify and speed optimize the code.
- Investigate the difference in speed if writing the code in the assembly language compared to C.
- Utilize hardware timers for each specific target if accuracy needs to be improved.
- Consider decreasing the user area size saving temporary information only for FOSA-processes in a container to minimize memory consumption (The container memory must then manually be freed).

One parameter that was never tested or measured is the effect that long list iterations would have on the system performance. In the two test cases only one and two FOSA-processes are created respectively meaning that no long list iteration will ever occur. Since the iteration will only be performed once per process the resulting iteration time is not expected to have decisive affect on the average performance. Still it should be tested and the effect should be evaluated.

The current implementation will as concluded in the previous section most likely to be sufficient for some applications. The extra latency caused by the measurement functionality will probably not cancel out the benefit of using spare capacity, unless the CPU is already highly utilized. The mean increase in context switching latency when swapping two FOSA-processes is 14us. This increase will lead to the switching time being 2% of the over all execution time at a swapping frequency of 1000 swaps per second instead of 0.6%. The worst case increase in service execution time is 1.4%, if the swapping frequency is less than 1000 swap/s the increase will be even less.

If the CPU utilization is initially low and a lot of spare capacity can be freed it is likely that execution time increase will be neglectable. The extra latency will only have a noticeable effect on systems where the amount of CPU spare capacity to distribute does not cover this extra latency. For a spare capacity distribution to be meaningful there must obviously be more capacity left to distribute than what would cover this extra latency. Conclusions that the measurement functionality can be used with the framework is drawn from assumptions that enough spare capacity can be freed for the contract based scheduling to be beneficial. These conclusions need to be reevaluated for each application.

Applications whose requirements on response time and latency are possible to meet under these conditions support the use of this functionality in the framework. Whether the extra memory consumption can be approved will depend on the available target memory space and on the memory requirements of the application. The additional latency depend on the system swapping frequency and the number of FOSA-processes on which execution time should be measured. In systems where the CPU utilization was initially low, where the number of FOSA-processes are few or where the swapping frequency is low, the extra latency is likely to go undetected.

The process execution time measurement functionality has been compiled together with the rest of FOSA by Erik Thorin. The implementation works as intended. What is left in terms of testing is to assure that this FOSA distribution will work with FRSH. Unfortunately the development of FRSH is behind schedule and not yet finished also delaying any such test. When these tests can be performed the measurements should again be evaluated and possibly improved. Finally it should be tested on different applications to find the exact latency and memory consumption tolerance levels for applications in different areas. Only then can it be concluded whether the measurement functionality is sufficient for general case applications.

As this thesis work is brought to an end the four initial objectives has been fulfilled. Solutions for measuring process execution times have been found, the solutions have been evaluated and the most suitable measurement method of those has been chosen for implementation. The implementation of this method has been performed and the functionality has been verified. Finally, test results have been analysed for performance evaluation.

List of Figures

2.1 Basic contents of an RTOS [33], [44]. 15

2.2 Common RTOS structure as depicted by Tom Sheppard [31]. 16

2.3 Process states and context switching [7]. 17

3.1 OSE memory configuration example [45]. 22

4.1 Modules in the FRESCOR framework [16]. 25

5.1 Connect to the RMM. 29

5.2 Kernel handler implementation flow chart. 30

7.1 Kernel handler and user area activation. 41

7.2 User area configuration. 42

7.3 Create handler flow chart. 43

7.4 Swap in handler flow chart. 45

7.5 Swap out handler flow chart. 46

7.6 A part of the ramlog print from a test performed in the soft core environment. 48

8.1 Test process structure and signal definition. 53

8.2 Part of the *run_ test()* function. 53

8.3 Illustration of test1 measurements. 54

8.4 Before modification of the *measure_ send_ swap_ receive* function. 55

8.5 After modification of the *measure_ send_ swap_ receive* fucntion. 55

8.6 The pBench *measure_ send_ swap_ receive* test. 55

8.7 Illustration of send+swap+receive measurement. 56

8.8 Time relative to number of swaps with and without execution time measurement. 58

9.1 Extra context switching latency due to measurements per second at different swapping frequencies. 63

List of Tables

6.1 Method evaluation table. 34

8.1 Benchmark pBench measurement results. 56

8.2 Measurement results from pBench using a FOSA process and kernel handlers. 57

8.3 Measurement results [ticks+us] making the figure 8.8 plot. 58

Bibliography

Litterature

[1] Jörn Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems — Thesis in fulfilment of engineering doctoral degree.* ISBN: 3-8322-1594-8. Aachen: Schaker Verlag, 2003.

[2] Ola Dahl. *Realtidsprogrammering.* ISBN: 91-44-03130-0. Lund: Författatren och Studentlitteratur, 2004. (In Swedish)

[3] Alan Burns and Andy Wellings. *Rel-Time Systems and Programming Languages — Chapter 13: Scheduling.* ISBN: 0-201-72988-1. Edinburgh: Pearson Education Limited, 2001 Third Edition.

[4] Brian Kernigan and Dennis Ritchie. *The C Programming language* ISBN: 0-13-110362-8. New Jersey: Prentice Hall, 1998 Second Edition.

[5] Avi Silberschatz, Peter Baer Glavin and Greg Gagne. *Operating System Concepts.* ISBN: 0471-69466-5. Hoboken: John Wiley & Sons Inc, 2005 Seventh Edition.

[6] Irv Englander. *The Archtechture of Computer Hardware ad Systems Software — An Information Technology Approach.* ISBN: 0471-31037-9. USA: John Wiley & Sons, Inc. 1996.

Manuals

[7] Enea Embedded Technology AB, 2006. *OSE Architecture User's Guide*

[8] Enea Embedded Technology AB, 2006. *OSE Core User's Guide*

[9] Enea Embedded Technology AB, 2006. *OSE Core Extensions User's Guide*

[10] Enea Embedded Technology AB, 2006. *OSE5.2 Getting Started*

[11] Enea Embedded Technology AB, 2006. *OSE Device Drivers User's Guide*

[12] Enea Embedded Technology AB, 2006. *OSE Illuminator User's Guide*

[13] Enea Embedded Technology AB, 2006.
OSE System Programming Interface Reference Manual

[14] Ismael Ripoll et al. 2006. *WP4: Execution Platforms Task: Software Quality Procedures.* FRESCOR Consortium.

[15] Ola Redell at Enea Embedded Technology, 2007 (9th revision). *OSE Development Policy.*

[16] Michael González Harbour, 2005 (Version 1.0). *Framework for Real-time Embedded System based on COntRACTS — Architecture and contract model for integrated resources.* FP6/2005/IST/5-034026 Deliverable: D-AC2v1. Universidad de Cantabria.

[17] Lauterbach Datentechnik GmbH, 1998. *Trace32 In-Circuit Debugger — Quick Installation Guide and Tutorial.*

[18] Alfons Crespo et al. 2007. *D-EP3v1 Implementation and Evaluation of the Processor Contract model 1.* FRESCOR Consortium.

[19] Enea Embedded Technology AB, 2006. *OSE Real-Time Operating System and Embedded System Development Environmentbitem*

[20] Freescale Semiconductor. *i.MX21 Reference Manual, Rev 2*

Web Pages

[21] The cygwin homepage, last viewed 25th of May 2007.
www.cygwin.com.

[22] The Tera Term homepage, last viewed 27th of June 2007.
http://hp.vector.co.jp/authors/VA002416/teraterm.html.

[23] The Lauterbach homepage, last viewed 27th of June 2007.
http://www.lauterbach.com/frames.html.

[24] The Blob homepage, last viewed 27th of June 2007.
http://www.lartmaker.nl/lartware/blob/.

[25] The FRESCOR project homepage, last viewed 23rd of July 2007.
http://www.frescor.org/.

[26] The FIRST project homepage, last viewed 23rd of July 2007.
http://130.243.78.209:8080/salsart/first/.

[27] The ENEA homepage, last viewed 7th of August 2007.
www.enea.com.

[28] Wikipegia about the POSIX standard, last viewed 7th of August 2007.
http://en.wikipedia.org/wiki/POSIX.

[29] Wikipedia about RTOSs, last viewed 8th of August 2007.
http://en.wikipedia.org/wiki/RTOS.

[30] Embedded system definitions, last viewed 27th of August 2007.
http://www.garggy.com/embedded.htm.

[31] Surreal-time RTOS webcourse by Tom Sheppard, last viewed 13th of July 2007.
http://www.surreal-time.com/Fundamentals/WBT/RTF_Free/ToC.html.

[32] The common man's guide to operating system design by Chris Smith, last viewed 13th of July 2007.
http://cdsmith.twu.net/professional/osdesign.html.

[33] RTOS course material by Ramon Serna Oliver at Keiserslautern Technical University. Last viewed 13th of July 2007.
http://www.eit.uni-kl.de/fohler/rt-course/material/RTOS.pdf.

[34] RTOS concepts, slides from slideshare by Sundar Resan. Lat viewed 13th of July 2007.
http://www.slideshare.net/sundaresan/rtos-concepts/.

[35] RTOS for DSPs, how to recover from deadlock. Last viewed 16th of July 2007.
http://www.dspdesignline.com/showArticle.jhtml?printableArticle=true&articleId=199400290

[36] Raquel S. Whittlesey-Harris presentation on real-time operating systems, last viewed 17th of July 2007.
http://deneb.cs.kent.edu/ mikhail/classes/es.u01/Realtime%20OS.ppt.

[37] A presentation on RTOSs by Lothar Thiele from the computer engineering and networks laboratory at the Swiss federal institute of technology. Last viewed 18th of July 2007.
http://lap.epfl.ch/advanced-courses/ThieleSep06_ RTOS_ Slides.pdf.

[38] Dr. Yair Amirs web cource lectures on operating systems. Last viewed 26th of July 2007.
http://www.cs.jhu.edu/ yairamir/cs418/600-418.html.

[39] Presentation on RTOSs by Prof. Dr. Antônio Augusto Fröhlich et al.
Last viewed 26th of July 2007.
http://www.lisha.ufsc.br/ guto/teaching/mpl/rtos.pdf.

[40] Presentation slides on OSE and RTOSs by Jan Lindblad, application engineer at ENEA, 7th December 1999 (In Swedish). Last viewed 7th of August 2007.
www.lysator.liu.se/upplysning/fa/RTOS.ppt.

[41] Subhashis Banerjee, Indian Institute of Technology, Delhi.
Last viewed 27th of August 2007.
www.cse.iitd.ernet.in/ suban/csl373/rtos.ppt.

Other

[42] E-mail contact with Magnus Karlsson, software engineer at the R&D OSE core labs. ENEA Stockholm.

[43] E-mail contact with Mathias Bergvall, embedded platforms contractor. ENEA Linköping.

[44] ENEA OSE Solution Guide (advertisement). *ENEA Embedded Automotive Platform — Building Automotive Applications with OSE.* 2005.

[45] ENEA OSE Real-Time Kernel datasheet (advertisement). *OSE Real-Time Kernel.* Available at www.enea.com, last viewed 29th of August 2007.

[46] Presentation slides by Eva Skoglund, Director of Product Marketing at ENEA, 2006. *OSE Values.* Available on the ENEA intranet.

Appendix A: **krn.con**

```
/* krn.con is the OSE5 static kernel configuration file. */

#ifdef USE_DEBUG
DEBUG          (YES)
#else
DEBUG          (NO)
#endif

POOL_SIZE      (0x280000)

ERROR_HANDLER  (sys_err_hnd)

/* Handlers, order is important for start handler 1's. */

START_HANDLER0     (bspStartHandler0)
START_HANDLER1     (bspStartHandler1)
VECTOR_HANDLER     (bspVectorHandler)
INT_MASK_HANDLER    (bspIntMaskHandler)
INT_CREATE_HANDLER  (bspIntCreateHandler)
INT_KILL_HANDLER    (bspIntKillHandler)
READ_TIMER_HANDLER  (bsp_read_timer)
CLEAR_TIMER_HANDLER (bsp_clear_timer)
#ifdef USE_POWER_SAVE
POWER_SAVE_HANDLER  (bspPowerSaveHandler)
#endif
CPU_HAL_HANDLER     (ose_cpu_hal_920t)
KERNEL_HALTED_HANDLER(ose_halted_hook)

#ifdef USE_RAMDUMP
START_HANDLER1      (ramdump_start_handler1)
#endif

/*Activate kernel handlers for process execution time measurement
 *xmlin 2007-06-15
 */
CREATE_HANDLER    (fosa_ose_create_handler)
SWAP_IN_HANDLER   (fosa_ose_swap_in_handler)
SWAP_OUT_HANDLER  (fosa_ose_swap_out_handler)
USER_AREA         (21)
```

Appendix B: fosa_ose_exe_time.c

```
/**
 * @file fosa_ose_exe_time.c
 * @brief: Using swap handlers to measure execution time for
 * individual ose processes. Time is measured in system ticks + a
 * number of micro seconds. The temporary slice start time is saved in
 * the process specific user area and the elapsed execution time of a
 * slice is added to the process specific list node in a shared memory linked list.
 *
 * $Author:   xmlin  $$Date:   2007-07-10
 *
 */

#include "ose.h"
#include "stdlib.h"
#include "malloc.h"
#include "ramlog.h"
#include "heapapi.h"
#include "ose_heap.h"
#include "string.h"
#include "node_info.h"



extern void delete_node(PROCESS pid);
extern PROCESS create_test_process(const char *name, OSENTRYPOINT *entrypoint
, OSPRIORITY priority);

fosa_ose_list_node_t *head_ptr = NULL;



/*Extend the memory storage area specific for each process. Ensures
 * fast acess of process data while inside kernel handlers. Configure
 * the "user area" statically. Add USER_AREA (<size>) to the krn.con
 * file.
 * - Process id
 * - Pointer to the list storage node for the current process
 * - Slice start time
 */
typedef struct user_area{
    OSTICK tick_slice_start_time;              /*unsigned long = 4 byte*/
    OSTICK micro_slice_start_time;             /*unsigned long = 4 byte*/
    fosa_ose_list_node_t *list_position_ptr;   /*pointer (4 byte ?)*/
    OSBOOLEAN fosa_process;                    /*size of unsigned char (1byte?)*/
    PROCESS pid;                               /*4byte*/
}user_area_t;



/*Create handler is called at process creation
 * - Set inital values to user_area variables
 */
void fosa_ose_create_handler(user_area_t *user_area_ptr, PROCESS id){

    if(head_ptr == NULL){
```

```
        /*Not a FOSA process if no nodes are created*/
        user_area_ptr->fosa_process = FALSE;
        return;
    }
    else{
        /*Possibly a FOSA Process*/
        user_area_ptr->fosa_process = TRUE;

        /*initialize user area*/
        user_area_ptr->pid = id;
        user_area_ptr->list_position_ptr = NULL;
        user_area_ptr->tick_slice_start_time = 0;
        user_area_ptr->micro_slice_start_time = 0;

        return;
    }
}



/*Swap in handler called at execution slice start.
 * -If first time in swap in, find the list node of interest.
 * -If node not found, not a fosa process.
 * -If node found, save a pointer to that node a the swap in
 *   time in the user_area.
 */
void fosa_ose_swap_in_handler(user_area_t *user_area_ptr){

    if(user_area_ptr->fosa_process == FALSE){
        return;
    }

    if(head_ptr == NULL){
        user_area_ptr->fosa_process = FALSE;
        return;
    }

    OSTICK ticks;
    OSTICK micros;
    ticks = get_systime(&micros);
    /*Save slice start time in user area*/
    user_area_ptr->tick_slice_start_time = ticks;
    user_area_ptr->micro_slice_start_time = micros;

    if(user_area_ptr->list_position_ptr != NULL){
        /*Not first time in swap in*/
        ramlog_printf("SWAP IN FOSA process %x\n", user_area_ptr->pid);
        return;
    }

    /*First time in swap in*/
    fosa_ose_list_node_t *tmp_node_ptr = head_ptr;


    while(tmp_node_ptr != NULL){

        if(tmp_node_ptr->pid == user_area_ptr->pid){

            ramlog_printf("First SWAP IN of FOSA process %x, NODE found\n",
            user_area_ptr->pid);

            user_area_ptr->list_position_ptr = tmp_node_ptr;
            user_area_ptr->fosa_process = TRUE;
```

```
            break;
        }

        if(tmp_node_ptr->next_ptr == NULL){
            ramlog_printf("NO node found with id: %x => IGNORE, not FOSA process\n",
             user_area_ptr->pid);
            user_area_ptr->fosa_process = FALSE;
        }
        tmp_node_ptr = tmp_node_ptr->next_ptr;
    }
}



/*Swap out called at a process slice ending.
 * - Calculate slice execution time
 * - Add slice time to previous execution slice times in the list node
 */
void fosa_ose_swap_out_handler(user_area_t *user_area_ptr){

    if(user_area_ptr->fosa_process == FALSE){
        return;
    }
    ramlog_printf("SWAP OUT FOSA process %x\n", user_area_ptr->pid);
    fosa_ose_list_node_t *node = user_area_ptr->list_position_ptr;

    OSTIME micros_per_tick = system_tick();
    OSTICK ticks, m;
    signed long micros;
    ticks = get_systime(&m);

    if(ticks < user_area_ptr->tick_slice_start_time){
        /*Tick Overflow*/
        ticks = ticks + micros_per_tick - user_area_ptr->tick_slice_start_time;
        user_area_ptr->tick_slice_start_time = 0;
    }

    /*Calculate time and add to node*/
    ticks = ticks - (user_area_ptr->tick_slice_start_time);
    micros = m - (user_area_ptr->micro_slice_start_time);
    node->nr_of_ticks += ticks;
    node->nr_of_micros += micros;

    while(node->nr_of_micros < 0){

        if(node->nr_of_micros > -micros_per_tick ){
            node->nr_of_ticks--;
            node->nr_of_micros = micros_per_tick + node->nr_of_micros;
            break;
        }
        else{
            node->nr_of_ticks--;
            node->nr_of_micros = micros_per_tick + node->nr_of_micros;
        }
    }
    while(node->nr_of_micros >= micros_per_tick){
        node->nr_of_ticks++;
        node->nr_of_micros = node->nr_of_micros - micros_per_tick;
    }
    return;
}
```

```
/*Delete node function.
 * - Redirect list pointers past the node.
 * - Free memory space for node.
 */
void delete_node(PROCESS pid){

    fosa_ose_list_node_t *tmp_node_ptr = NULL;

    if(head_ptr != NULL)
        tmp_node_ptr = head_ptr;
    else if(head_ptr == NULL){
        ramlog_printf("ERROR: tried to delete node from not existing list for %x\n", pid);
        return;
    }

    for(;;){
        if(tmp_node_ptr->pid == pid){
            /*Remove last node in list*/
            if(tmp_node_ptr->next_ptr == NULL && tmp_node_ptr->prev_ptr != NULL){
                (tmp_node_ptr->prev_ptr)->next_ptr = NULL;
            }
            /*Remove first node in list*/
            else if(tmp_node_ptr->next_ptr != NULL && tmp_node_ptr->prev_ptr == NULL){
                head_ptr = tmp_node_ptr->next_ptr;
                (tmp_node_ptr->next_ptr)->prev_ptr = NULL;
            }
            /*Remove from the middle of the list*/
            else if(tmp_node_ptr->next_ptr != NULL && tmp_node_ptr->prev_ptr != NULL){
                (tmp_node_ptr->prev_ptr)->next_ptr =
                 tmp_node_ptr->next_ptr;

                (tmp_node_ptr->next_ptr)->prev_ptr =
                 tmp_node_ptr->prev_ptr;
            }
            else
                head_ptr = NULL;

            heap_free_shared(tmp_node_ptr);
            break;
        }

        if(tmp_node_ptr->next_ptr != NULL)
            tmp_node_ptr = tmp_node_ptr->next_ptr;
        else{
            ramlog_printf("ERROR: tired to delete not existing node for %x \n", pid);
            break;
        }
    }
}


/*Create FOSA process function, overloading create_process()
 * - Allocate memory for node and add node to list.
 * - Set initial values to the node.
 * - Create process
 */
PROCESS create_test_process(const char *name, OSENTRYPOINT *entrypoint, OSPRIORITY priority){

    fosa_ose_list_node_t *mynode = (fosa_ose_list_node_t*)
    heap_alloc_shared(sizeof(fosa_ose_list_node_t), __FILE__,
    __LINE__);
```

```
    /*initialize node*/
    mynode->next_ptr = NULL;
    mynode->prev_ptr = NULL;
    mynode->nr_of_ticks = 0;
    mynode->nr_of_micros = 0;
    mynode->pid = 0x0;

    if(head_ptr == NULL){
        head_ptr = mynode;
    }
    else{
        head_ptr->prev_ptr = mynode;
        mynode->next_ptr = head_ptr;
        head_ptr = mynode;
    }

    PROCESS mypid_= create_process(OS_PRI_PROC, name, entrypoint, 200,
    priority, 0, 0, NULL, 0 , 0);

    mynode->pid = mypid_;
    ramlog_printf("FOSA process %x and list node created.\n", mynode->pid);
    start(mypid_);
    return mypid_;
}
```

Appendix C: **test_app.c**

```
/*
 * @file test_app.c
 * @brief: Test application to find measurement overhead.
 * - Main test process starting test series.
 * - Test processes to swap between.
 * - Actual test function to run.
 *
 * $Author:   xmlin  $$Date:   2007-07-10
 */


#include "ose.h"
#include "stdlib.h"
#include "ramlog.h"
#include "heapapi.h"
#include "ose_heap.h"
#include "node_info.h"
#include "malloc.h"
#include "string.h"
#include "stdio.h"

extern PROCESS test_application_main_;
extern fosa_ose_list_node_t *head_ptr;
OSBOOLEAN with_nodes;

#define NR_OF_COUNTS 10
#define WITH_NODES TRUE

#define SWAP_SIGNAL 1000

struct Swap_signal{
    SIGSELECT sig_no;
    PROCESS p0;
    PROCESS p1;
    unsigned long count;
};
union SIGNAL{
    SIGSELECT sig_no;
    struct Swap_signal swpsig;
};

static const SIGSELECT sel_swap[] = { 1, SWAP_SIGNAL};
void run_test(int points);



/*Test process0*/
OS_PROCESS(process0){

    union SIGNAL *sig;
    while(1){
        sig = receive(sel_swap);
        sig->swpsig.count--;
        if(sig->swpsig.count == 0){
            send(&sig, test_application_main_);
```

```
            kill_proc(current_process());
        }
        else
            send(&sig, sig->swpsig.p1);
    }

}

/*Test process1*/
OS_PROCESS(process1){

    union SIGNAL *sig;
    while(1){
        sig = receive(sel_swap);
        sig->swpsig.count--;
        if(sig->swpsig.count == 0){
            send(&sig, test_application_main_);
            kill_proc(current_process());
        }
        else
            send(&sig, sig->swpsig.p0);
    }
}

/*Test process notfosa*/
OS_PROCESS(notfosa){

    kill_proc(current_process());
}



/*Main test process
 * -Set test parameters
 * -start run test function
 */
OS_PROCESS(test_application_main){

    /*printf("ej Klar\n");
    delay(5000);*/

    int i;
    for (i = 1; i < 2; ++i) {
        run_test(i);
    }

    printf("Klar\n");
    while(1) {
        delay(10000);
    }
}



/*
 * Run test function
 * -Create test processes
 * -Wait for test processes to finish
 * -Delete allocated list node memory
 * -Calculate and print elapsed test time
 */
void run_test(int points){
```

```
OSTICK t1, t2;
OSTICK m1, m2;
t1 = get_systime(&m1);
OSTIME mpt = system_tick();

PROCESS p0 = 0x0, p1 = 0x0;

ramlog_printf("---------------------------Run Test----------------------\n");
/*ramlog_printf("Node size = %d\n", sizeof(fosa_ose_list_node_t));*/


if(WITH_NODES){
    p0 = create_test_process("Process0", process0, 0);
    p1 = create_test_process("Process1", process1, 0);
}

if(!WITH_NODES){
    p0 = create_process(OS_PRI_PROC, "Process0", process0, 200, 0, 0, 0, NULL, 0 , 0);
    /*attach(NULL, p0);*/
    start(p0);
    p1 = create_process(OS_PRI_PROC, "Process1", process1, 200, 0, 0, 0, NULL, 0 , 0);
    /*attach(NULL, p1);*/
    start(p1);
}


PROCESS pNotFOSA = create_process(OS_PRI_PROC, "pNotFOSA", notfosa, 200, 0, 0, 0, NULL, 0 , 0);
ramlog_printf("Process %x of NOT FOSA type created\n", pNotFOSA);
start(pNotFOSA);

static const SIGSELECT sel_swap[] = { 1, SWAP_SIGNAL};
union SIGNAL *sig1;
sig1 = alloc(sizeof(struct Swap_signal), SWAP_SIGNAL);
sig1->swpsig.p0 = p0;
sig1->swpsig.p1 = p1;
sig1->swpsig.count = points*NR_OF_COUNTS;
send(&sig1,p0);
sig1 = receive(sel_swap);
if(sig1->swpsig.count != 0)
    ramlog_printf("count != 0\n");
free_buf(&sig1);

if(WITH_NODES){
    /*fosa_ose_list_node_t *tmp = head_ptr;
    while(tmp != NULL){
        ramlog_printf("ID %x ticks %d micros %d\n",
        tmp->pid, tmp->nr_of_ticks, tmp->nr_of_micros);

        tmp = tmp->next_ptr;
    }*/

    delete_node(p0);
    delete_node(p1);
}


t2 = get_systime(&m2);
signed long m = m2-m1;

/*Calculate elapsed test time*/
if(t2 < t1){
    /*Overflow*/
    t2 = t2 + mpt - t1;
```

```
        t1 = 0;
    }
    OSTICK t = t2-t1;
    if(m < 0){
        while(m < -mpt){
            m = mpt + m;
            t--;
        }
        if(m < 0 && m > -mpt){
            m = mpt + m;
            t--;
        }
    }
    while(m >= mpt ){
        m = m - mpt ;
        t++;
    }
    /*ramlog_printf("%d %4d.%4d\t #Count ticks,micros mpt = %d\n", points*NR_OF_COUNTS, t, m, mpt);*/
    ramlog_printf("---------------------------Test END---------------------\n");
    return;
}
```

Appendix D: node_info.h

```
/**
 * @file node_info.h
 * @brief: Node structure for the list of process execution times
 *
 * $Author:   xmlin  $$Date:   2007-07-10
 *
 */



#ifndef NODEINFO_
#define NODEINFO_

void delete_node(PROCESS pid);
PROCESS create_test_process(const char *name, OSENTRYPOINT *entrypoint
, OSPRIORITY priority);


/*Node structure for the list:
 * -Process id and elapsed execution time, ticks and microseconds
 * -pointer to the next and previous node in the list
 *Dynamically linked list structure for storage of execution time.
 */
typedef struct listnode{

    PROCESS pid;
    OSTICK nr_of_ticks;
    signed long nr_of_micros;

    struct listnode *next_ptr;

}fosa_ose_list_node_t;

/*Create an empty header node to be the first node in the list*/
extern fosa_ose_list_node_t *head_ptr;

#endif /*NODEINFO_*/
```

Appendix E: **osemain.con**

```
/* osemain.con fragment for exeTm processApp - start. */

PRI_PROC( test_application_main, test_application_main, 1000, 0, DEFAULT, 0, NULL )

/* osemain.con fragment for exeTm processApp - end. */
```

Appendix F: **Makefile**

```
COMPANY = Enea Embedded Technology AB
VERSION = 1.0
LIBDESC = An execution time measurement for OSE processes

TARGETS := lib

include ../../environment.mk

#object files for library
LIBOBJECTS += fosa_ose_exe_time.o test_app.o

#Path to source code for $(LIBOBJECTS)
vpath %.c $(REFSYSROOT)/modules/fosa_ose_exe_time/src

include $(REFSYSROOT)/modules/modules.mk
```

Appendix G: fosa_ose_exe_time.mk

```
#osemain.con fragment for exe_time
OSEMAINCON += $(REFSYSROOT)/modules/fosa_ose_exe_time/src/osemain.con

#Hello library to include in kernel link module and load modules
LIBS += $(REFSYSROOT)/modules/fosa_ose_exe_time/$(LIBDIR)/libfosa_ose_exe_time.a

#Define needed by krn.con and board.c-files
DEFINE += -DMODULES_FOSA_OSE_EXE_TIME
```